

**Software Tools User Group  
USENIX Association  
/usr/group**

**UNICOM Conference Proceedings**

**January 1983  
San Diego, CA**

---

**Software Tools User Group  
USENIX Association  
/usr/group**

**UNICOM Conference Proceedings**

**January 1983  
San Diego, CA**

---

© 1983 by Software Tools User Group, The USENIX Association and /usr/group

All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain  
with the author or author's employer.

UNIX is a trademark of Bell Laboratories.

Other trademarks noted in the text.

---

## Preface

The 1983 Winter UNICOM meeting was held from Tuesday, January 25 to Friday, January 28, 1983, in San Diego, California. UNICOM was a joint meeting of the Software Tools Users Group, the USENIX Association and /usr/group to share information about UNIX and UNIX-like Operating Systems, the C language, and the Software Tools.

Tom Strong, Executive Director of USENIX and editor of the USENIX newsletter *;login:*, with the aid of other reporters, attended most sessions and wrote summaries of the talks given. These scribe notes appear in *;login:* and Software Tools Communications and were reproduced in these Proceedings. The Editors wish to thank Tom Strong, Mark Bartelt, F. Arlene Spurlock, Scott Thompson, James Perskey, Sam Penny, and Jeffrey C. Bruce for their efforts in preparing these notes.

The Proceedings is comprised of the scribe notes, abstracts and complete texts of those papers submitted for publication. Sam Penny and F. Arlene Spurlock of Penny, Penny and Strong managed the production of this volume on behalf of the Software Tools Users Group.

The papers and notes are arranged in order, by session, as presented. An author index and permuted keyword-in-context index of titles is provided in the back of the book.

For more information on any of the groups, write to the addresses below. Contact the Software Tools Users Group for information regarding additional copies of these Proceedings.

Software Tools User Group, Inc.  
1259 El Camino Real # 242  
Menlo Park, CA 94025

USENIX Association  
P.O. Box 7  
El Cerrito, CA 94530

/usr/group  
P.O. Box 8570  
Stanford, CA 94305-0221



---

**This page intentionally left blank**

---

# CONTENTS

## Software Tools User Group Meeting

<b>Users Group Status Report .....</b>	<b>1</b>
Dave Martin	
<b>Rockies Association for Tools (RAFT) Report .....</b>	<b>1</b>
Bill Meine	
<b>Software Tools in C? .....</b>	<b>3</b>
Phil Scherrer	
<b>Enhancements to <i>format</i> .....</b>	<b>7</b>
Bob Calland	
<b>A Portable Mail System for the Software Tools .....</b>	<b>7</b>
Joe Sventek	
<b>Interactive Data Analysis using the Software Tools .....</b>	<b>8</b>
Van Jacobson	
<b>New Tools for the Virtual Operating System .....</b>	<b>9</b>
Bob Upshaw and Van Jacobson	
<b>Update on Software Tools Implementation .....</b>	<b>14</b>
John Henshaw	
<b>Tools in Australia .....</b>	<b>14</b>
Paul Hausman	
<b>LISP for the Software Tools VOS .....</b>	<b>15</b>
Charlie Dolan and Dave Martin	
<b>West Coast Implementors Group Proposed Standards .....</b>	<b>15</b>
Bob Upshaw, Joe Sventek, and Van Jacobson	

## W1 — Welcoming Remarks and Keynote Address

<b>Welcome .....</b>	<b>17</b>
Tom Uter and Bill Appelbe	
<b>Software Army on the March .....</b>	<b>17</b>
John Mashey	

---

## W2 — News from . . .

<b>/usr/group</b> .....	22
Bob Marsh	
<b>USENIX</b> .....	22
Lou Katz	
<b>Usenet</b> .....	22
Mark Horton	
<b>AT&amp;T</b> .....	23
Bob Guffy	
<b>/usr/group — Standards Committee</b> .....	23
Heinz Lycklama	
<b>UC Berkeley</b> .....	23
Robert Fabry	
<b>Sun Microsystems</b> .....	24
Bill Joy	
<b>DEC</b> .....	24
Bill Munson	

## W3A — Performance Enhancement and Measurement

<b>UNIX System III and 4.1BSD</b> .....	25
John Chambers and John Quarterman	
<b>Contiguous Load Modules for UNIX</b> .....	39
Steve Zucker	
<b>Improved Schedulers for Non-Paged UNIX Systems</b> .....	39
Darwyn Peachey	
<b>An Implementation of the <i>vfork</i> System Call for PDP-11 UNIX</b> .....	40
Michael Karels	
<b>Handling Very Large Programs on a 16-Bit Super-micro</b> .....	41
Mitch Bishop	

## W3B — UNIX System V

<b>System V Offering</b> .....	48
W. R. Guffy	
<b>System V Support Offering</b> .....	48
Dave Sandel	
<b>Licensing Activity and Pricing</b> .....	49
Larry K. Isley	

---

## W4A — Networks

<b>The Plexus Networked UNIX .....</b>	<b>51</b>
Monte Pickard	
<b>CSNET Status Report .....</b>	<b>51</b>
G. Brendan Reilley	
<b>Mail Systems and Addressing in 4.2BSD .....</b>	<b>53</b>
Eric Allman	

## W4B — Development Tools

<b>BIBFIND — A Bibliographic Retrieval System .....</b>	<b>63</b>
James A. Moyer	
<b>COBOL Compiler Construction Experiences Using <i>lex</i> and <i>yacc</i> .....</b>	<b>69</b>
Robert E. Conant and Herbert G. Mayer	
<b>Development of <i>refer</i>: Bug Fixes and Enhancements .....</b>	<b>99</b>
Bill Tuthill	
<b>RAPID: A Tool for Building Interactive .....</b>	<b>105</b>
Anthony I. Wasserman and David T. Shewmake	

## T1B — UNIX in the Office Environment

<b>A Uniform and Simple User Interface to UNIX .....</b>	<b>113</b>
Spencer Rugaber	
<b>UNIX Time-sharing Menu-driven Office System .....</b>	<b>116</b>
James A. Neyer	
<b>A Friendly Text Processing Environment .....</b>	<b>116</b>
Arthur Zemon	
<b>Writing User Documentation for UNIX Systems .....</b>	<b>117</b>
Jean Yates and Rebecca Thomas	

## T2B — New UNIX Implementations I

<b>Hewlett-Packard's Entry into the UNIX Community .....</b>	<b>119</b>
Frederick W. Clegg	
<b>4.2BSD on the Sun Workstation .....</b>	<b>132</b>
Tom Lyon and Bill Shannon	
<b>Experiences in Porting 4.1BSD UNIX to the <math>\lambda</math>750 VLSI .....</b>	<b>132</b>
Paul Chen and Chet Britten	

---

<b>UNIX on Apollo Computers .....</b>	<b>133</b>
Eric R. Shienbrood, Carl A. Soeder, James R. Ward and Kincade N. Webb	

## **T3A — Programming Languages and Environments**

<b>Towards a UNIX System Ada Programming Support Environment .....</b>	<b>143</b>
J. Eli Lamb	
<b>LINUS (Leading Into Noticeable UNIX Security) .....</b>	<b>143</b>
Steven M. Kramer	
<b>UNIX Logo .....</b>	<b>145</b>
Brian Harvey	
<b>A Global Optimizing C Compiler .....</b>	<b>151</b>
George Powers	

## **T3B — Novel Applications of UNIX**

<b>UNIX Research at Lucasfilms .....</b>	<b>167</b>
Jim Lawson	
<b>ARIEL: An Experimental UNIX-based Interactive Video Information System ..</b>	<b>167</b>
R. C. Haight and D. B. Knudsen	
<b>Development of a Digital Simulation System .....</b>	<b>169</b>
William Raves and James Cassidy	
<b>Meeting the Coming UNIX Training Challenge .....</b>	<b>177</b>
Jay R. Hosler	

## **T4A — Technical Reports**

<b>UNIX for the STD Bus .....</b>	<b>185</b>
Luigi Cerofolini	
<b>Ctrace - A Portable Debugger for C Programs .....</b>	<b>187</b>
J. L. Steffen	
<b>VAX11 Compatibility on PDP-11s .....</b>	<b>193</b>
Donn Seeley	
<b>The IS/1 Workbench for VAX/VMS .....</b>	<b>199</b>
William Torcaso	

---

## **T4B — Database Systems**

<b>Research Database Management Software for UNIX-based Microcomputers .....</b>	<b>201</b>
F. W. Stitt	
<b>Design &amp; Implementation of the DB Relational Database Management System</b>	<b>211</b>
J. Robert Ward	
<b>Interactive System/Three and the Intel Data Base Processor .....</b>	<b>229</b>
John R. Levine	
<b>/rdb: A Relational Data Base Management System .....</b>	<b>237</b>
Rod Manis	
<b>Focus/USE: A Low Keystroke Database Editor .....</b>	<b>241</b>
Anthony I. Wasserman, Martin Kersten	
<b>The Informix Commercial DBMS for UNIX .....</b>	<b>245</b>
Laura L. King	

## **F1A — Graphics**

<b>Device Independent Graphics Enhancements at ITTDCD .....</b>	<b>247</b>
Greg Hidley	
<b>Terminal-Independent Plotting Packages .....</b>	<b>251</b>
Don Mackay	
<b>Graphics Standards for Personal Workstations .....</b>	<b>257</b>
Michael Shantz	
<b>Windows with 4.2BSD .....</b>	<b>260</b>
Steven R. Evans	
<b>Computer Animation at UCSD .....</b>	<b>261</b>
Jeff Loomis and Phil Mercurio	

## **F1B — New UNIX Implementations II**

<b>REGULUS, a Real-Time UNIX Lookalike .....</b>	<b>268</b>
Bill Allen	
<b>UNIX for the National 16032 .....</b>	<b>269</b>
Paul Neelands, Richard Miller and Chris Sturgess	
<b>The Port of UNIX to the Gould 32/27 .....</b>	<b>273</b>
Jack Blevins	
<b>A Menu-Driven Real-Time UNIX System .....</b>	<b>279</b>
Kent Blackett	
<b>EUNICE .....</b>	<b>284</b>
Ellen Williams	

---

## **F2A — New UNIX Implementations III**

<b>Porting UNIX .....</b>	<b>285</b>
P. Verbaeten and Y. Berbers	
<b>UNIX on the National Semiconductor NS16032 .....</b>	<b>291</b>
Glenn C. Skinner and Bill Jolitz	
<b>UNIX for the Computer Automation 4/95 .....</b>	<b>307</b>
Steve Pozgaj	
<b>Architectural Implications of UNIX .....</b>	<b>307</b>
Matt Dickey, Greg Noel, Bob Querido, Bill Appelbe and Jim McGinness	

## **F2B — Marketing and Venture Capital**

<b>Getting Venture Capital .....</b>	<b>308</b>
Henry Wilder	
<b>UNIX Markets and Competition .....</b>	<b>308</b>
Bob Katsive	
<b>Delivering UNIX to the End-User Market .....</b>	<b>311</b>
Michael Denny	
<b>Distribution and Differentiation .....</b>	<b>313</b>
Marlene Martin	
<b>New UNIX Markets in Engineering .....</b>	<b>313</b>
Camran Elahian	

## **F3A — Portability**

<b>Portability in the UNIX World .....</b>	<b>314</b>
Mike O'Dell	
<b>A Tutorial on C Portability .....</b>	<b>315</b>
Michael Tilson	
<b>IS/3: A Compatible Extension of UNIX System III .....</b>	<b>325</b>
Steven Zucker	

## **F3B — Languages and Programming Environments**

<b>VMS C Compiler .....</b>	<b>330</b>
Jean Wood	
<b>UNIX APL .....</b>	<b>330</b>
Joseph Yao	

---

<b>The NIAL Language Project .....</b>	<b>331</b>
M. A. Jenkins	
<b>SOLID: for On-Line Systems Development .....</b>	<b>333</b>
John R. Mashey	

## F4 — Standardization

<b>The /usr/group Standards Committee .....</b>	<b>335</b>
Heinz Lycklama	
<b>The History and Purpose of Standards .....</b>	<b>348</b>
Eric Petersen	
<b>Standards Organization: Levels and Measurement .....</b>	<b>348</b>
Jim Isaak	
<b>Criteria for Standards .....</b>	<b>349</b>
Robert Swartz	



---

**This page intentionally left blank**

## Users Group Status Report

*Dave Martin*

Software Tools Users Group Coordinator  
Hughes Aircraft Co.  
Bldg. R1, Mail Stop C320  
P.O. Box 92426  
Los Angeles, CA 90009

*Tapes:* The general tapes essentially have not changed in two years. Some machine-specific tapes have more and/or newer tools (e.g., *mail*, *tcs*). There was a report that the Real Time Systems Group (RTSG) at Lawrence Berkeley Laboratory (LBL) is working on a new general tape. A release form for contributors is being developed.

The tapes were available in several formats at the meeting. The tape formatted for Univac machines was not available then. An overseas distributor for the tapes is needed so mailing costs can be reduced.

The tapes are in the public domain so everyone should feel free to re-distribute them. Anyone can order a tape and submit it to a distributing group. The Fall-82 release of the RSX-11/VAX tapes are on the DECUS Symposium tapes.

*Group Organization:* The Board has been pursuing incorporation of STUG.

There are three categories of membership: inexpensive; industrial (about \$150) for companies deriving benefit from the group and/or the tools; and sustaining member organization for companies that provide significant services, funds, supplies, and/or equipment to STUG. Current sustaining member organizations are Lawrence Berkeley Laboratory, Hughes Aircraft Company, and Direct, Incorporated. A variety of things like tapes, a low-cost printer, etc., are needed. Companies with an interest in the tools are invited to discuss sustaining membership with members of the Board.

The Real Time Systems Group at LBL has become quite active in STUG recently, including producing the newsletter immediately before the meeting. (However, it will not be able to produce any more newsletters.) The editorial "we" in that newsletter was the four-member STUG Board.

## Rockies Association for Tools (RAFT) Report

*Bill Meine*

Louisiana Land and Exploration Company  
3900 South Wadsworth Blvd.  
Suite 660  
Lakewood, CO 80235

RAFT consists of about twelve members with about six different Software Tools implementations. The members get together every four to six weeks. They have found and shared corrections for bugs and some unwieldy tools.

---

The RAFT members started with the STUG distribution tape. They send a tape around the to the members of the group and the current tape holder uses *ics* to modify its contents. When the next member receives the tape the tools at that site are updated to the version on the tape. This way one person has the "standard" at a time and eventually all sites are brought up to date with the changes made at the other sites. In addition, changes made for one machine can be tested for portability and usefulness by members with other machines. One trip around the loop takes about three months. They have found *ics* to be very helpful in maintaining records of what changes have been made and why.

Ben Domenico of the National Center for Atmospheric Research (NCAR), another active member of RAFT, commented that they had found a need to make and signify changes by context rather than by line numbers so concurrent multiple changes to a module could be resolved more easily. They have not found a solution to this problem.

---

## Software Tools in C?

*Phil Scherrer*

Carousel MicroTools, Inc.  
609 Kearney Street  
El Cerrito, CA 94530

The Software Tools in ratfor have been widely accepted by sites with a FORTRAN orientation. They have proven to be quite portable to the machines in such sites. However as the C language has become more widely available there has been growing interest in converting the ratfor version of the tools to C. There are at least two reasons why systems that have C would need the Software Tools.

1. Many of the systems that have C compilers do not have the carefully designed and well tested utilities and library that are available with the Software Tools. However the tools are written in ratfor, which needs FORTRAN, and on many microprocessor systems a good C compiler is available long before a viable FORTRAN compiler.
2. C is more a powerful language than ratfor for development. It would be very nice to have the tools available on these new systems, and for many people and new systems it is undesirable to wait for a FORTRAN compiler to have access to the tools.

There are (at least) two ways that the ratfor tools could be made usable on C-oriented machines. The first would be to make a ratC compiler which would produce C instead of FORTRAN. The code generated would resemble ratfor more than C and some of the features of C like *struct* would not be used, which would lead to some inefficiencies. A C version of the library would have to be defined which would have the same calling conventions, functions, and names as the STUG library. This solution would maintain the compatibility with STUG while allowing new work to be done in either C or ratfor.

The second way would be to do a one-time machine-aided translation to true, full C, including converting all library routines to their UNIX counterparts. This would allow bringing the tools code to UNIX binary license systems with full efficiency in C. However, it would break the connection with STUG.

Carousel MicroTools is working on the first solution — ratC — to bring its Software Tools on CP/M to other micro-based systems. Carousel will propose a set of C calling conventions for the new STUG library when it is finally agreed upon.

There were some comments from the audience:

- What about going to Pascal instead of C? Pascal is not as well standardized as C and is not as widely available or efficient.
- Keep the primitive set but do translate the tools to C.
- Do not compromise the primitives or the tools if they are going to be translated to C.
- There is a version of the tools in the public domain that can be run — and changed — on binary UNIX systems.
- Whitesmiths is reportedly writing a Fortran-to-C translator; it could be used to translate *ratfor* output.

---

## Software Tools in C ?

Philip H. Scherrer  
Carousel MicroTools, Inc.  
609 Kearney St.  
El Cerrito, CA 94530

The Software Tools in RATFOR have found wide acceptance by programmers forced to live in a FORTRAN dominated environment. The STUG version of the tools and library have proved to be readily portable to a large number of operating systems, bringing much of the utility and the Unix style of simplicity to those systems. Now there is growing interest in converting the RATFOR versions of the tools to the C language. One might wonder why the RATFOR tools would be needed on systems with support for C. There are at least two compelling reasons.

First, many microcomputers are using operating systems without a large carefully designed set of utility programs with supporting libraries. The STUG version of the Tools is a natural candidate to fill this void. The Tools however are written in RATFOR which needs FORTRAN. When the tools movement started, FORTRAN was the logical choice for a host "machine language" since it was universally available. This is no longer true. The development of quality affordable FORTRAN compilers seems to be lagging the availability of good C compilers by several years for newer microprocessor operating systems. There is therefore a clear need for the tools in a C environment on a large variety of machines.

Second, C is a more powerful (and more dangerous) language to use for future development work. It would be very nice to have the STUG tools and library to build upon when making new tools on non-Unix systems. (One can argue that some of the tools style of simplicity that grew from the original Unix versions actually provides a more friendly environment than Unix for many programmers). This provides an independent reason to consider the conversion to C.

What are the choices and what is lost? One can list several paths to tools in C:

### How Often?

- A. One time hand translation.  
This is a large error prone task.
- B. Macro-aided semi-automatic translation.  
Still labor intensive and subject to the more difficult to find errors of hand translation.
- C. Automatic translation - ratC compiler  
A large program, comparable to the current RATFOR preprocessor in complexity with some cases still needing hand corrections.

---

What Form?

A. Translate to "Nice" C.

This would mean using C-like parameter passing rather than FORTRAN-like where only pointers are actually (usually) passed. Data structures should be incorporated for clarity in many cases.

1. Modified STUG library conventions.  
i.e. keep the STUG names and specifications for the library.
2. Unix library conventions.  
Adopt the stdio package and "chapter 2" system calls rather than the STUG primitives.

B. Translate to "Ugly" C treating C as a machine language not to be viewed by humans.

1. Maintain present STUG library and pass everything as pointers.
2. Define C version of the STUG library to have the same functional specifications but C-like calling conventions.

At least two useful solutions can be picked from the above map. The first is to build a RatC compiler which would produce C readable code. The library calling conventions would be those of the present library so that new RATFOR programs could be readily utilized by those systems with C but not FORTRAN. A C version of the library would be defined which would have the same functions and names as the STUG library, but fewer ampersands needed on the calling side. It would be expected that new work could be done in RATFOR or in C. This would keep the connection with STUG while allowing new development to go on in C. This is the path that is being taken by Carousel MicroTools in its quest to bring the tools to the underprivileged.

Another solution is to do a one-time machine aided translation to true C. This would include converting all library routines to their Unix counterparts and re-coding those parts of the code that are clumsy with the new library specifications. The purpose of this would be to bring the tools base of code to Unix binary license systems for future modifications and new tool development.

There are advantages and disadvantages to both possible solutions. In the first case, the code produced will resemble ratfor more than C, and will not be able to take advantage of many of the code efficiencies of the C language. On the other hand, the tie to STUG

---

will be maintained so new developments can easily move to and from the C based systems. In the second case this connection will be lost since the exchange could be only in one direction. But perhaps in a short time the people who are now coding in RATFOR will be coding in C anyway.

As part of its interest in this project, Carousel MicroTools will prepare a proposed set of C calling conventions for the new STUG library when it is finally agreed upon.

---

## Enhancements to *format*

**Bob Calland**

Naval Ocean Systems Center  
Code 621 (B)  
San Diego, CA 94152

Mr. Calland described a new formatting tool. It is based on the *format* tool described in *Software Tools* but has the appearance and much of the power of UNIX's *nroff*.

This tool offers many features that a *format* user might find useful — even necessary — to produce sophisticated formats. The most important of these features are IF-ELSE constructions, string and number registers (both useful for making more general macros), and the ability to specify distances in a variety of units (needed, for example, for outputting to devices with different numbers of characters per inch).

An *nroff/troff* user will recognize that this tool is based on the documentation for those programs, although the commands are not quite the same. In general the differences seem to be improvements to *nroff*. Automatic hyphenation is based on an IBM 360 program; Knuth's methods will be implemented.

The tool was written in ratfor but it does not use the Software Tools primitives. It does not handle sophisticated printers yet. There is no plan to produce a device-independent file as it is intended for utility documents.

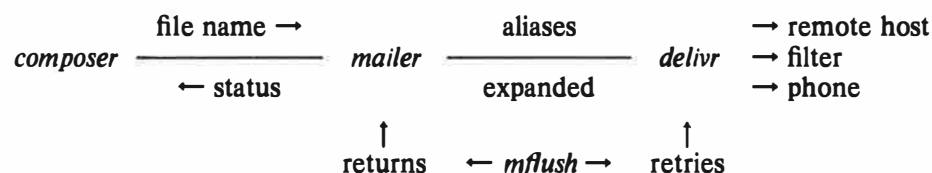
STUG has been given a copy of the program and may have it converted to use the primitives.

## A Portable Mail System for the Software Tools Virtual Operating System

**Joe Sventek**

Lawrence Berkeley Laboratory  
Computer Science and Mathematics Dept.  
Berkeley, CA 94720

Mr. Sventek has developed a portable mail system. This was done by partitioning mail into user-interface modules and a delivery system. The user-interface is produced locally. The mail from the composer is placed in a "posting slot" for the delivery system. The delivery system communicates with other delivery systems, if necessary, and places the mail in an appropriate place for the recipient's mail interface. The basic layout is:



*Mflush* is used to clean up for *mailer* and *deliver*.

A number of assumptions were made about portability:

1. the tools VOS is available;



2. the specification *~directory/file* is understood by *open* and *create*;
3. the directory primitives listed in the CACM paper on the VOS are available; and
4. five additional primitives are defined.

The protocol used by the *delivr* routine for delivery is the same as for Arpanet. The current Arpanet standards are followed; these assume a reliable, flow-controlled circuit.

The current implementation used *ratnet*, which used *mmdf* [?] for packet protocol. *Ratnet* was written by Dave Martin and is being debugged by Mr. Sventek.

A "white pages name server" which will look up the path to a mail recipient is planned. (For example, it would answer the question "how do I get this mail to the recipient named usenix?") Also planned is a "receipt requested" feature.

## Interactive Data Analysis using the Software Tools

*Van Jacobson*

Lawrence Berkeley Laboratory  
Real Time Systems Group, Building 46A  
Berkeley, CA 94720

The Real Time Systems Group at Lawrence Berkeley Laboratory supports about 200 systems with about eighty people. It has standardized on the Software Tools for internal use.

They were recently asked to support a problem of exploratory data analysis and decided to use the tools. They found that they needed tools for four general functions.

Data presentation: two new tools called *graph* and *tbl* were created.

Data transformation: a new tool called *mkhist* was created.

Data manipulation: the data was often corrupted or had noise or other extraneous stuff. The existing tools were found to be satisfactory for their needs.

data generation: They used the existing tools plus a new one called *record*.

The tools proved to be very satisfactory for this project. It was very easy for the professionals to learn and use them. Several examples of how the tools were used were presented.

A total of about fourteen tools were added for data analysis. They had to change or add to *dc*, *field*, *sort* and *uniq*. In several commands they removed "pretty print" so the output could be used in pipes.

The tools and changes developed will be offered for the next distribution tape. All are done in *ratfor* except part of the *graph* tool.

---

## New Tools for the Virtual Operating System

*Bob Upshaw*

*Van Jacobson*

Lawrence Berkeley Laboratory  
Real Time Systems Group, Building 46A  
Berkeley, CA 94720

This talk described development of Software Tools versions of the UNIX tools *yacc* and *lex*.

*Lex* is a "lexical analyzer" or "scanner." The input to *lex* is a series of regular expressions that define "tokens" and the actions (in a language such as C or ratfor) to be performed when these tokens are found. The output is a scanner routine written in the language of implementation. *Yacc* (it stands for Yet Another Compiler-Compiler on UNIX) is a "parser generator." It generates functions that take tokens provided by a lexical analyzer and organizes them according to specified rules. When a rule has been recognized a user-provided action is performed.

A public domain portable parser generator called LR, which had been developed at Lawrence Livermore Laboratory, was obtained. They found that LR was portable and did work but had several shortcomings that had been solved in UNIX *yacc*. They designed a set of tools to overcome the shortcomings. Their *yacc* takes *yacc-like* input and generates ratfor output. They have used it to build a significantly enhanced *dc* and are using it to develop a new, upward compatible, C shell-like shell.

They have also developed a UNIX-like *lex* tool that generates ratfor code. The input is similar to that of UNIX *lex* except it follows the Software Tools standard regular expression syntax. Their *lex* has been used to develop a tool to look at source files and determine the language they are written, for writing scanners for some simple languages, and for the scanner for the new shell mentioned above.

They plan to distribute these tools only through the Software Tools Users Group. Mr. Upshaw commented that when the proposed Software Tools standards are resolved they will make their tools conform.

---

# New Tools for the Virtual Operating System

*Bob Upshaw*

*Van Jacobson*

Real Time Systems Group  
Lawrence Berkeley Laboratory  
University of California  
Berkeley, California 94720

The Real Time Systems Group at Lawrence Berkeley Laboratory has spent over four years developing software tools for use on our various machines. There are many good reasons for putting the Software Tools on a computer system, but our main one is the desire to increase productivity in the areas of program development, data handling, and data analysis. We chose the Software Tools because they were UNIX-like and UNIX [4] is well known for improving productivity. (For just a few of many references on this topic, see [5].)

Unfortunately, not every machine runs UNIX (yet), but every machine can (and many do) run the Software Tools. So, in striving to build ourselves better tools, we often look to UNIX for guidelines and to the Virtual Operating System for a vehicle. This paper describes a few of the tools we have developed or are in the process of developing. Naturally, these tools are designed to run in the Virtual Operating System environment and therefore are (should be) portable. Any similarities between these tools and UNIX tools are purely intentional; we do not intend to re-invent the wheel.

## 1. YACC

In the May, 1981 issue of *IEEE Transactions on Software Engineering* there is a paper describing a *portable* LR(1) parser generator[1]. The parser generator (named 'LR') was developed at Lawrence Livermore Labs and is in the public domain. Naturally, we got our hands on it and performed some experiments. It is portable and does indeed work, but suffers from the shortcomings of most parser generators, including

- The inability to assign token values to terminal symbols, making it difficult to write a scanner which would work regardless of the changes to the grammar.
- The inability to attach an action (semantics) to a production, making it necessary to modify the code generation section of a compiler whenever the slightest changes were made to the grammar.
- The requirement to order recursive productions in an awkward way to force the desired associativity.
- The requirement to write the grammar productions in an awkward way in order to specify precedence or to avoid ambiguities in the parser (such as shift-reduce conflicts.)

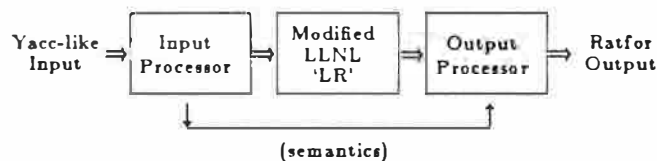
Yet, despite LR's shortcomings it represented an impressive program. Since the above problems are solved by Yacc[2], we decided to make LR work like Yacc. Hoping to avoid

---

code modifications to LR as much as possible, we designed a set of tools which did the following:

- Took a Yacc-like input grammar and split it into two pieces:
  - 1) A BNF grammar to be processed by LR.
  - 2) A file containing the semantics associated with each grammar production.
- Took the output of LR (the parse table) and the semantics generated by the first tool and put them together to form a ratfor program — the desired compiler.

Here is a picture of how our Yacc tool works:



We have since used our Yacc tool to build a significantly enhanced dc tool, and we are using it to develop a new shell.

## 2. LEX

Using the dragon book[3] as a reference, we have implemented a Lex tool similar to the UNIX Lex tool (except in ratfor.) For those who may not be familiar with Lex, it is to scanners what Yacc is to parsers. The input to Lex specifies regular expressions which describe the tokens recognized by a lexical analyzer (i.e. a scanner) and the actions to be performed when those tokens are recognized in the input stream. The output of Lex is the scanner written in ratfor.

There is still more work to be done on our Lex, mostly in the area of performance optimization, but we have already used it to generate a 'typeof' tool (which looks at a source file and determines its language), scanners for some simple languages, and a scanner for our new shell (below).

As with Yacc, our Lex input is similar to the UNIX Lex input except we followed the tools standard regular expression syntax (sigh). The conversion of a UNIX-style Lex input specification to a tools-style Lex input specification is straight forward, thus allowing the use of existing Lex input files which are in the public domain.

---

### 3. SHELL

We are currently working on a new shell specification which is to be upwards compatible with the current VOS shell. Our current spec contains control structures, variables, procedures (a mechanism to replace 'alias' as suggested by Bill Joy in the csh manual), a 'history' mechanism, wildcards, command substitution, and so forth. Most of what we have done has very much the flavor of the C-shell. Our 'sh' spec has been frozen and we have made a first pass on the grammar and semantics.

### 4. PRINTF

Yes, Virginia, there is a C-like 'printf', 'fprintf', and 'prints' written in ratfor. Our routines are *almost* portable and will probably work on almost all the machines running the VOS. The only rule we violate is passing data of one type and declaring it as another. However, this should work almost everywhere since we are very careful not to *use* a data item except in routines where it is declared correctly.

### 5. PEEK

We are writing a portable tool similar to UNIX 'more' which we are calling 'peek'. ('More' is a UNIX tool which works like 'crt' but uses raw-io and allows additional commands following each prompt.) Peek is not named 'more' because the commands to peek are slightly different than those of more. Peek will allow the user to jump forwards or backwards in a file or a group of files, search forwards or backwards for a regular expression, etc.

### 6. Conclusion

The purpose of this note was to make the members of STUG aware of what we are doing in the hopes that they will

- not duplicate our efforts needlessly,
- tell us what they are doing so we will not duplicate their efforts,
- and give us suggestions or feedback when they believe we may be on the wrong track.

Any correspondence is welcome and can be addressed to the authors at:

Real Time Systems Group, Build. 46A  
Lawrence Berkeley Labs  
1 Cyclotron Rd.  
Berkeley, Ca. 94720

By the way, this paper was produced completely by the use of the Software Tools. Our newest project is a tool which filters 'roff' input and produces 'TeX' [7] input, which is then run thru a TeX formatter. Thus, our roff documents can be output to a phototypesetter-quality printer, as was done here.

---

## 7. References

- [1] "LR — Automatic Parser Generator and LR(1) Parser", Charles Wetherell and Alfred Shannon. *IEEE Transactions on Software Engineering*, May, 1981.
- [2] "YACC — Yet Another Compiler-Compiler". Stephen C. Johnson, Bell Labs.
- [3] *Principles of Compiler Design*, Aho and Ullman, Addison-Wesley.
- [4] UNIX is, as we all know by now, a trademark of Bell Labs.
- [5] Even the vendors who used to deny the existence of UNIX now admit it's strengths:  
  
DEC: "VNX Plan Bridges VAX/VMS, UNIX Environments", *Insight*, Vol. 2, Number 8, October, 1982.  
  
AMDAHL: "UNIX On Mainframes: Amdahl", *commUNIXations*, August, 1982. Page 13.  
  
IBM: "How Data Flow Can Improve Application Development Productivity", *IBM Systems Journal*, Vol. 21, No. 2, 1982. Page 167-178.
- [6] "A Parser Generator for a 'Production' Programming Environment", Theresa Breckon, 1982. (To be published.)
- [7] *T<sub>E</sub>X and METAFont — New Directions in Typesetting*, Donald E. Knuth, American Mathematical Society and Digital Press.

---

# Update on Software Tools Implementation Data General's RDOS

*John Henshaw*

CompuCode  
6147 Aspenwall Road  
Oakland, CA 94611

At the last Software Tools Users Group meeting Mr. Henshaw reported on his implementation of the Software Tools on Data General machines running the RDOS operating system. In this talk he discussed some things he has done to improve the efficiency of his implementation. Two general notions were mentioned.

1. When the operating system provides primitives such as a "putline" it can be more efficient to consider them as tools primitives instead of having the tools *putlin* call *putch* for each character.
2. Tiny, often-used tools such as *echo* can be put in the shell as subroutine calls to eliminate the loading time for the tool. [On systems with slow disk access this can make a difference in the user's perspective of the tools.]

## Tools in Australia

*Paul Hausman*

This talk described experiences with the tools in a site with a large FORTRAN base. They changed ratfor to produce FORTRAN-77 output so they could use the "char" data type. After some time they moved back to ratfor (using "int" for characters) because it took so much work to maintain separate ratfor and rat77 libraries. They had also run into other problems trying to use FORTRAN-77.

Mr. Hausman expressed a strong desire for extensions to ratfor for use in applications (not necessarily in the tools themselves). Ratfor has changed their way of working. The tools have proven useful and have been well accepted. They found them to be a reasonable alternative to UNIX for their FORTRAN-oriented users.

---

# LISP for the Software Tools VOS

*Charlie Dolan and Dave Martin*

Hughes Aircraft Co.  
PO Box 92426  
Los Angeles, CA 90009

This talk described an implementation of LISP for the Software Tools.

[Unfortunately I missed this talk and there was no abstract or paper available.]

## West Coast Implementors Group Proposed Standards

*Bob Upshaw, Joe Sventek, and Van Jacobson*

Lawrence Berkeley Laboratory  
Berkeley, CA 94720

This presentation was based on the premise that since we do share the Software Tools we need standards. As the tools and the tools usage have grown the need for certain changes and additions to the standards have become apparent. The problem with changing a standard is to balance the need for improvements against the changes that will be forced on people using code that conforms to the current standard. Additionally there are the needs to define how code meeting the current standard is to be affected and to plan an orderly method of distribution of changed code.

### *Ratfor*

Mr. Sventek presented some proposed changes for ratfor:

1. a `-n` flag to prevent loading of the definitions file;
2. groups of FORTRAN statements be permitted in the initialization and re-initialization clauses of "for" statements, with a syntax the same as in C;
3. an initial legal op value in the "arith" macro;
4. enhanced conditional pre-processing;
5. character literals of the form 'c' or '@c' be legal; and
6. long variable names no longer be legal.

Mr Sventek has a *ratfix* program that converts ""-delimited strings to '"'-delimited strings, old conditionals to new ones, and long names to short ones.

He proposed another change for the future: a post-processor to convert string literals to character arrays and re-order FORTRAN declarative statements to ANSI-66 FORTRAN. This would require a change to *remark*.

He also proposed site-specific tailoring be allowed: long names, FORTRAN-77 character strings, and "alpha\_characters" — strings of characters that are legal in addition to the default forms.

Manual pages with the proposed changes are available from

Joe Sventek  
Mail Stop 50B-3238  
Lawrence Berkeley Laboratory  
Berkeley, CA 94720



---

### *Primitives*

Mr. Jacobson proposed the following changes to the primitives:

1. add a "child.error" status return to *spawn*;
2. add an "OK / error" argument to *endst*;
3. add a status return to *prompt* that would return what *getlin* returned;
4. add more escape sequences to *esc*;
5. add "tagged patterns" to *match* [and other routines that I did not record].

He also proposed a new tools primitive manual.

### *Extensions*

Mr. Upshaw commented that they did not have any extensions to propose beyond those proposed at the Boston meeting. There was to be a meeting that evening for all interested parties to discuss the proposals and make a decision. The audience was asked if anyone present had any opposition to the people present making decisions on a new standard; no one objected.

Bill Plauser was asked if he had any plans to update the *Software Tools* book; he said no.

The meeting ended with plans to meet that evening for a Board meeting and discussion and adoption of a modified standard.

## WELCOME

*Tom Uter and Bill Appelbe*

University of California at San Diego  
La Jolla, CA 92093

These gentlemen, who were respectively the general chairperson and technical co-chairperson for UNICOM, welcomed the very large crowd. Over 100 abstracts had been submitted; 60–70 were accepted.

## KEYNOTE ADDRESS

### Software Army on the March — Project Strategies and Tactics

*John Mashey*

Bell Laboratories  
Whippany, NJ 07981

Mr. Mashey developed an analogy between software development and an army campaign to explain and help validate the concept of “fast prototyping” as a project management tool. The domain under discussion was significant software and/or hardware projects in internal industrial efforts but not necessarily research or government environments.

Project planning was presented as involving three stages:

- strategies — what to do
- tactics — how to do it
- implementation — do it

The talk focused on the first of these.

No preference was expressed for any specific project lifecycle description. Decisions have to be made on which methodology to use and then how and when to allocate resources.

An abstract model was presented as a game graph. There are many ways to get from the starting point to the end, which was represented as a plane rather than a single point [like going from New York City to somewhere in California]. Routes (arcs) lead to nodes where alternate arcs can be taken, often leading to other major arcs. Some arcs, indeed some major arcs, can lead to dead ends or in unreasonable directions or require resources that are not available. The problem is to understand the attributes of the game and find a way to select the arcs to be taken. The attributes of the game:

- decisions are multi-staged, like chess
- there are many people involved — friends and opponents — and you sometimes cannot tell what others are doing

- there is often incomplete or hidden information
- the game is not completely deterministic
- the rules may vary over time
- the universe may change.

When a node is reached there are several things to consider when deciding which arc to take:

- there is a cost associated with each move
- it takes time to move
- there is risk, both in the present and the future, associated with the choice of arcs
- the capacity, both in the present and the future, required by choosing an arc
- the cost, time, risk, and capacity required to unmake a move.

Two limit-case node configurations were described:

- a single path from one node to another [like a single bridge over a river] — take care and spend a lot of time analyzing the move
- many paths from one node to another — take the easy one as you can always redo it later

An analysis of the six considerations for selecting an arc (cost, time, initial and continuing risk and capacity) will enable you to recognize the alternatives and tradeoffs so you can make informed decisions.

Decisions should be sorted by impact: identify those with large ripple effects, prefer those that preserve alternatives, and defer those that can be helped by later information. Different configurations of arcs and nodes (e.g., parallel or sequential arcs) will affect the process.

Mr. Mashey went on to present a concrete analogy between an army marching and a large software project. The major points he made are summarized below.

*Running the project:* define the project goal; do the development; maintenance; and completion.

*While running the project you need to be aware of:* budgets and schedules; the capabilities of the personnel; and requirements, both political and for the end-user.

*The types of personnel involved in a project include:* systems engineers to help the technical design; human performance engineers to help the human interface design; people who can produce reliable, maintainable production code; people, including new persons on the project, who can do the less demanding coding; fast prototypers who can, quickly and at low cost, find and try out new and/or innovative and/or risky solutions; support and tools groups to provide a usable starting base and the tools needed for development; research groups; upper management, with whom there must be two-way communication; and others, including friends and skeptics of the project and people who are threatened by it.

*Project resources required include:* existing and planned tools, including old but proven solutions (it is difficult to build a new set of tools in the middle of a project); strategic planning; and existing knowledge of the domain.

*Project strategy involves:*

- identifying existing components that fit into the project;
- identifying and analyzing the components that are needed and their alternatives;
- establishing a firm starting base;
- identifying the areas of highest risk or uncertainty and starting the fast prototypers on them as soon as possible; (this often involves pursuing parallel arcs); and
- involving the skeptics and end-users as early as possible.

A project can fail in spite of the best planning. It may have been the wrong solution to the problem or hardware or software limitations or that someone came up with a better solution. When that happens management must know how to recognize it and be prepared to cut their losses.

---

## **Software Army on the March — Project Strategies and Tactics**

John R. Mashey  
Bell Laboratories  
Whippany, NJ 07981

This talk describes the work of an army building roads ("software projects") through a part of the countryside that:

- seldom has current maps.
- is plagued by earthquakes ("major environment changes"), flash floods ("temporary problems"), and fog ("inability to predict the future").
- contains villages of natives ("users") who may greet roadbuilders with anything from great interest to outright hostility, but whose cooperation is essential.
- may be overrun by enemy forces ("competitors") trying to build their own roads to the same locations.

An effective campaign has two aspects:

- Doing the right thing, i.e., fighting the right war in the right place, and choosing good routes to reach the goals.
- Doing it right, i.e., building maintainable roads with adequate capacity, at a reasonable cost, without losing too many casualties, and without offending the natives.

Most software methodologies emphasize the second part; this talk emphasizes the first by examining decision processes and methods of analyzing routes. Two different viewpoints are used. The first is the formal game theory viewpoint -- making decisions in a nondeterministic, multi-stage, N-person, non-zero-sum game played with incomplete information. The second is the "army" model described above. From this viewpoint will be discussed such issues as:

- The need for scouts on motorcycles ("fast prototypers").
- How campaigns differ, and thus affect choice of troops, ranging from commando raids through the march of the hordes.
- Special precautions for earthquake territory.
- Getting natives to buy and drive your trucks, instead of shooting your tires out as you drive through their villages.

There exist many similarities in the decision processes of formal game analysis, military planning, and project management. This talk uses the first and second to help shed light on the third. Provided with the talk is an annotated bibliography, which includes a famous treatise on "project management" written in 500 B.C.

An overhead projector is required for this talk.

---

Software Army on the March --  
Project Strategies and Tactics

John R. Mashey  
Bell Laboratories  
Whippany, NJ 07981

- BLA78A Blake, S.P., Managing for Responsive Research and Development, W. H. Freeman, San Francisco, 1978.  
Chapter 4 (strategies, risk analysis) especially good: alpha strategy (careful frontend analysis) vs beta (prototyping, adaptation to state-of-the-art changes). Example of Sidewinder development (simple, reliable, cost-effective U.S. air-to-air missile) good example of beta strategy.
- BUS65A Busacker, R.G., Saaty, T.L., Finite Graphs and Networks: An Introduction With Applications, McGraw-Hill, New York, 1965.  
See applications to games and puzzles in sections 6-8 - 6-13.
- CLA51A Clarke, A. C., Superiority. Across the Sea of Stars, Harcourt, Brace, New York, 1959.  
A humorous science-fiction story of war lost by the side with better technology, because they kept changing it.
- DUF80A Duffy, N.D., Assad, M.G., Information Management - An Executive Approach, Oxford University Press, Cape Town, South Africa, 1980.  
Chapter 2 (Information Systems Development Life Cycles) characterizes strategies as Linear, Loopy Linear, Plug-in, and Prototype, then offers attributes that may guide choice of strategy for a given project.
- DUN80A Dunnigan, J. F. The Complete Wargames Handbook. Morrow, New York, 1980.  
Survey of serious games, which often resembles real life in aspects of strategic planning, allocation of scarce resources, and risk analysis.
- FAL81A Fallows, J. National Defense, Random House, NY, 1981.  
Superb analysis of US military weapons procurement and realities. Chapters: Realities, Managers, Magicians, Two Weapons, Employees, Theologians, Changes. "Two Weapons" chapter: F-16 fighter, brilliantly designed with good analysis and prototyping; then made less capable by adding things in opposite direction of original design philosophy.
- GOM82A Gomaa, H., The Impact of Rapid Prototyping on Specifying User Requirements, Proc. ACM SIGSOFT Software Engineering Symposium: Rapid Prototyping, Columbia, Md, April 19-21, 1982, paper #14.  
Case study of project done at GE.
- JON80A Jones, A.J., Game Theory: Mathematical Models of Conflict, Wiley, New York, 1980.  
Chapter 1 is reasonable introduction to fundamental concepts and analysis techniques (decision trees, minimax, etc).
- KEE81A Keen, P.G.W., Information Systems and Organizational Change. Comm. ACM 24, 1 (Jan 1981), 24-33.  
Discusses long-term change in organizations related to information systems. Reviews causes of social inertia, resistance, and counter-implementation. Good reading for any software designer who hopes to sell ideas. Includes good bibliography.

- LEH80A Lehman, M. M. Programs, Life Cycles, and Laws of Software Evolution. Proc. IEEE, 68 9 (Sept. 1980), 1060-1076.  
S-, P-, and E-programs; use of methods to predict release effects.
- LUC57A Luce, R.D., Raiffa, H., Games and Decisions - Introduction and Critical Survey, Wiley, New York, 1957.  
Chapters 1,3,4,7,8 form a reasonable introduction to the topic.
- MAC68A Macksey, M.C. Panzer Division - The Mailed Fist. Ballantine Books, New York, 1968.  
History of German tanks in WW II.  
Moral: when you're even or ahead, don't stop, somebody is gaining on you. When you're behind, be careful how you try to catch up.
- MAR82A Martin, J. Application Development Without Programmers. Prentice Hall, Englewood Cliffs, NJ, 1982.  
Illustrates existing methods and tools for 10X - 100X productivity improvements over conventional methods, for some problem domains. In some cases, normal high-level language coding appears to be pathetically obsolete.
- PYS82A Pyster, A., Boehm, B. W., The Impact of Rapid Prototyping on Software Development Standards. Proc. ACM SIGSOFT Software Engineering Symposium: Rapid Prototyping, Columbia, Md, April 19-21, 1982, #28.  
TRW builds Software Productivity System on UNIX with fast prototyping.
- TAY82A Taylor, T., Standish, T. A., Initial Thoughts on Rapid Prototyping Techniques, Proc. ACM SIGSOFT Software Engineering Symposium: Rapid Prototyping, Columbia, Md, April 19-21, 1982, paper #40.  
Some parts are a bit academic, but has some good overall thoughts on prototyping, especially the section "Limits of Prototyping" on page 13.
- TZUXXA Tzu, Sun. The Art of War. 500 B.C. Military Service Publishing Company, Harrisburg, PA, 1944. [thanks to L. Bernstein]  
Contains numerous pithy discussions of project management, although not expressed in the standard terminology.  
``When you engage in actual fighting, if victory is long in coming, them men's weapons will grow dull and their ardour will be dampened.... Thus, though we have heard of stupid haste in war, cleverness has never been associated with long delays.''  
  
``[Frederick the Great, in his Instructions to his Generals, says "Those generals who have had but little experience attempt to protect every point; while those who are better acquainted with their profession, guard against decisive blows at decisive points, and acquiesce in smaller misfortunes to avoid greater." In other words, keep away from sideshows.]''
- WEI82A Weiser, M., Scale Models and Rapid Prototyping, Proc. ACM SIGSOFT Software Engineering Symposium: Rapid Prototyping, Columbia, Md, April 19-21, 1982, paper #42.  
Offers clear characterization of 3 different kinds of prototypes: user interface, functionality, and performance.

Chairperson: *Armando Stettner*  
DEC

**/usr/group**  
*Bob Marsh*

Mr. Marsh welcomed the audience and then presented seven items.

1. /usr/group plans to change its name to uniFORUM to better reflect the purposes of the group.
2. The terms of several directors are up in June. A nominating committee has been formed. Mr. Marsh will not be running for another term as president.
3. The draft standard for UNIX is seen as the most important, highest priority item for the group. [See the previous issue of *;login:* for more information on the draft standard...Ed.]
4. A local chapter startup kit is available from the group.
5. The newsletter, *commUNIXcations*, will be featuring issues devoted to specific topics such as word processing on UNIX.
6. Corporations may join the group as sponsors. (Normal membership is available to individuals only.)
7. The /usr/group Board decided in October to put on its own conference annually. The Board felt that they had different goals and objectives that could not be met at a joint conference with USENIX. However, said Mr. Marsh, this may not be the best thing and the decision was being reconsidered. A decision was promised by Friday of UNICOM. [See the announcement in the last issue of *;login:*...Ed.]

**USENIX**  
*Lou Katz*

Mr. Katz stated the purposes of the USENIX Association and told of the efforts of the new Association office to support those purposes.

The future of joint conferences was under discussion within the USENIX Board and with members of the /usr/group Board. The main question to be resolved was what was best for the UNIX community. An answer to the question of future joint conferences with /usr/group was to be announced by Friday.

There was to be a Board meeting with the membership on Wednesday. The Board was reviewing how to make the tutorials more available, as there had been tremendous oversubscription of some of them.

*Mark Horton*

Mr. Horton made the distinction between *usenet* and the *uucp* network: *usenet* is that collection of sites who receive the newsgroup called "net.general." There was to be a Birds—of—a—Feather that night to provide information on how to join *usenet*.

---

**AT&T**  
*Bob Guffy*

Mr. Guffy told us AT&T was going to announce System V. An information packet on System V was to be available on the vendor tables. He reported that UNICOM was the largest gathering of people with an interest in UNIX.

## **/usr/group — Standards Committee** *Heinz Lycklama*

Mr. Lycklama introduced the draft UNIX system interface standard that is being developed by the /usr/group Standards Committee. It is not a UNIX standard; rather it is a system interface standard for end users and applications writers. It is based on System III. [The draft standard was discussed in the last issue of *login*; it may be obtained for \$50 from /usr/group, P.O. Box 8570, Stanford, CA 94305...Ed.]

## **UC Berkeley** *Robert Fabry*

Dr. Fabry announced that 4.2BSD is “almost” ready. 4.2BSD is the latest version of UC Berkeley’s paging UNIX for the VAX<sup>†</sup>. In 4.2BSD, neither the system nor applications are limited by address space. It will be available to all UNIX/32V and System III licensees. Binary licenses will be available from several sources (not UCB). It was developed with financial support from ARPA, manufacturers, and industry. The software is in the public domain except for that covered by AT&T’s proprietary interests.

Features of 4.2BSD are: Ethernet\* support with 1.2 Mbit per second end-to-end throughput on an 11/750; Arpanet support; a file system that is 4–10 times faster than that on 4.1BSD; new methods of interprocess communication; new symbolic debugger, new *sendmail*; faster *f77*; a worm shell (for local networks); file mapping in address space; support for the VAX 11/730 and new peripherals; and isolation of VAX specifics.

4.2BSD is about 53,500 lines of code or about 46% bigger than 4.1BSD. Of this about 4800 lines is VAX-specific C code and about 1100 lines is VAX-specific assembler code. Large multi-user systems with lots of devices and networks use about .5mb main memory for resident code and buffers. Such systems should have at least 2mb of main memory.

4.2BSD is expected to be available in May. Current 4BSD licensees will receive information as the time approaches. For more information about 4.2BSD write to:

Pauline Schwartz  
Computer Systems Research Group  
Computer Science Division, EECS  
University of California  
Berkeley, CA 94720

The paper “Hints on Configuring a VAX” has been revised and is available from the same address.

---

<sup>†</sup> VAX is a trademark of Digital Equipment Corporation.

\* Ethernet is a trademark of Xerox Corporation.



---

## Sun Microsystems

*Bill Joy*

Mr. Joy was one of the primary people responsible for the Berkeley Software Distributions. He has recently begun working at Sun Microsystems while helping to finish 4.2BSD.

Mr. Joy sees two *de facto* standards for UNIX: 4.2BSD and System V. These, he suggested, support two different user communities: those whose needs are similar to those of the ARPA community (very large programs, virtual memory requirements, extensive networking, etc.), and those with more traditional timesharing applications who need to support more users and business and computer center operations.

He suggested that a common base system interface standard be developed so applications software could be run on either system.

## DEC

*Bill Munson*

Mr. Munson announced that "DEC supports UNIX" across its full product line. DEC strongly supports the /usr/group system interface standard and will implement it in VMS/VNX for "commercial use."

DEC has announced a C compiler for their Professional computer and is considering offering V7 for it. They will be offering binary licenses for UNIX V7 on PDP<sup>†</sup> machines with their V7M-11 product during the summer of 1983. 4.1BSD will be offered for "technical use" late in 1983. On their 36-bit machines they offer the University of Utah Version 7 port that runs under TOPS-10.

---

<sup>†</sup> PDP is a trademark of Digital Equipment Corporation.

Chairperson: *Terrence Miller*  
Hewlett Packard Co.

## UNIX System III and 4.1BSD; a Practical Comparison

*John Chambers*

University of Texas Medical Branch  
Galveston, TX 77550

*John Quarterman*

University of Texas at Austin  
Austin, TX 78712

A practical comparison of System III and 4.1BSD stemming from production experience with both systems on the same hardware (a VAX-11/780) may be useful to those deciding which to use, especially as the educational license for System III has only recently been made available, and 4.2BSD may not be suitable for many installations. Previous discussions of System III have emphasized its origins, rather than directly comparing it with 4.1BSD.

This paper compares the two systems in several areas, including: initial installation, booting, and configuration; languages, shells, typesetting, graphics, source code control, and data base; terminal handler (*fcntl*, *ioctl*, KMC-11 support) and other device drivers; operations, maintenance, and robustness; and games.

---

## UNIX\* System III and 4.1bsd

### A Practical Comparison

John Chambers

Office of Academic Computing & Biostatistics  
University of Texas Medical Branch, Galveston

John Quarterman

Computation Center  
University of Texas at Austin

### ABSTRACT

A practical comparison of System III and 4.1bsd stemming from production experience with both systems on the same hardware (a VAX-11/780\*\*) may be useful to those deciding which to use, especially since the educational license for System III has become available only recently and since 4.2bsd may not be suitable for many installations. Previous discussions of System III have emphasized its origins, rather than directly comparing it with 4.1bsd.

This paper compares the two systems in several areas, including: initial installation, booting, and configuration; languages, shells, typesetting, graphics, source code control, and data base; terminal handler (*fcntl*, *ioctl*, KMC-11 support) and other device drivers; operations, maintenance, and robustness; and, of course, games.

#### 1. Introduction

This paper describes certain differences between System III and 4.1bsd, leaving details of common functions to the manuals. This is a *qualitative* comparison, intended to serve only as a guide for further study. Both systems are considered as operating systems for a VAX-11/780, though occasional references are made to System III and 2.8bsd on a PDP-11\*\*.

While we are aware that many installations would prefer to run System V or 4.2bsd as soon as they become available, we have not had access to sufficient information to adequately compare these two systems. Also, it is likely that there will still be those who wish to run System III or 4.1bsd until the newer systems are proven reliable.

The sections below roughly follow the sections of Volume I of the Unix Programmer's Manual.

Presumably most of the obvious bugs to which we allude have been fixed in the successor systems.

\* UNIX and System III are Trademarks of Bell Laboratories.

\*\* VAX and PDP are Trademarks of Digital Equipment Corporation.

## 2. Installation and Configuration

The installation and configuration documents are sufficiently complete that few problems should be encountered when following their instructions. Known problems are noted below.

### 2.1 Installation

Both systems are delivered in the traditional Unix format, viz. a set of half inch magnetic tapes containing copies of all the binaries, source code, and documentation, plus accompanying hardcopy documentation (Bell sells manuals ready for use, while Berkeley supplies duplication-quality masters).

System III binary licenses are available, and the Berkeley distribution is also available on two RK07 disk packs.

For those unfamiliar with VAXen, "Installing and Operating 4.1bsd" contains sections on VAX hardware terminology and disk formatting which have no counterparts in "Setting Up UNIX" (the System III installation guide). The formatting section (actually a pointer to *format(8)*) saves time that would otherwise be spent finding the same information in the VAX diagnostics handbook. It is to be noted, however, that a stand-alone RP06 formatter is in fact available on the System III release and is perfectly usable ... if you happen to find it.

#### 2.1.1 System III boot bugs

The tape bootstrap printed in *vaxops(8)* is wrong. The last instruction should be "S0", not "S 2". The on-line copy (/stand/conflp/mt0boo.cmd) is correct, however.

The startup code in /unix is also wrong:

```
diff start.s.orig start.s
14,15c14,19
< mtptr $Scbbase,$SCBB # set SCBB
< mtptr $_Sysmap,$SBR # set SBR
---
> # Both SCBB and SBR should be loaded with physical, not virtual,
> # addresses, so the region bits should be zero. VAX-11/780 Release 7
> # doesn't care for SBR, but causes a reserved/privileged instruction
> # trap if the SCBB region is non-zero.
> mtptr $Scbbase-0x80000000,$SCBB # set SCBB
> mtptr $_Sysmap-0x80000000,$SBR # set SBR
```

Both the SCBB and SBR registers require physical addresses, but the code attempts to load them with system space addresses, i.e. with the high bit set. On VAXen before Revision 7, released in the summer of 1982, the microcode apparently failed to check for this. On a Rev 7 VAX the "mtptr SCBB" instruction causes a privileged instruction trap; as it happens, the "mtptr SBR" still works.

To bring up System III /unix, attempt to boot it and let it trap, thus loading it in memory. Then either load the SCBB register manually and start at the instruction following the "mtptr SCBB", or patch the "mtptr SCBB" value and start over.

Needless to say, changing this code and rebuilding the kernel object should have high priority in the installation process.

We assume these boot bugs are fixed in System V.

We had no such problems bringing 4.1bsd up in an afternoon.

## 2.2 Configuration

Both systems are relatively easy to configure.

System III is limited in the number and variety of peripherals for which drivers are supplied; this is not true of 4.1bsd, which has a wide variety of drivers. Only the RP06 disk controller is supported by System III on a VAX-11/780, whereas 4.1bsd supports many controllers, including the RK07, RM05, RM80, etc.

Further, 4.1bsd understands the full interconnection architecture of the VAX, so that it is possible to have, say, two RP06's on one MASSBUS and another on a second; all RP06's must be on the same MASSBUS in System III. This flexibility of 4.1bsd extends to allowing the system to decide at configuration time which MASSBUS's the RP06's are on.

Rumor has it that System V supports most of the devices that System III lacked. However, it apparently still does not have the auto-configuration ability of 4.1bsd.

## 3. Commands

The general utility commands supplied with the two systems exhibit relatively minor differences, mainly in terms of options. A few commands are included in each distribution which do not occur in the other; many of these are of questionable usefulness, and the reader is referred to the manuals for further details. Certain of the larger packages, however, such as language support facilities, are rather different, and are discussed in the following sections.

### 3.1 Languages

System III has extended *make* to include many additional default rules to handle common conditions, to the point that many compilations require no makefile. Additions have also been made to handle archives and SCCS files (see below), and to make use of environment variables and defaults. The 4.1bsd *make* seems to be very much in the flavor of V.7.

The two systems have almost indistinguishable versions of the C compiler, except for the Berkeley addition of very long identifiers (decried in some quarters as a violation of the language standard). The *as* assembler, the *ld* linker, and associated libraries are similar. Both compiler optimizers have (different) bugs related to floating point computations that make them unusable in production.

Both systems support *f77*. 4.1bsd has some bug fixes and other improvements (an overlaid version of this compiler is available for 2.8bsd). Both also support Ratfor and the Extended Fortran Language (EFL), but 4.1 additionally provides

the *struct* utility, used to convert FORTRAN sources into reasonably clean Ratfor.

System III has *bs*, essentially derived from BASIC. There is no equivalent in 4.1bsd.

Conversely, 4.1 includes PASCAL, (Franz) Lisp, and APL, none of which are supported in System III.

### 3.2 Shells

System III supports the Bourne shell (*sh*), with few noticeable changes from V7. 4.1bsd has much the same Bourne shell plus the Cshell (*csh*), often the new user's first command language.

The Cshell has most of the capabilities of the Bourne shell (though the syntax is different), plus the history and alias features. These two features (which allow replaying, editing, and saving commands) are the main reason many users prefer the Cshell (although some cite its control structures as another reason).

The 4.1 Cshell also has a set of job control features (requiring the new Berkeley terminal driver) which allow the user to suspend and resume sub-processes.

### 3.3 Formatting and Typesetting

4.1 offers the -me macro package, while System III has the -mm package, somewhat augmented from PWB. 4.1 includes but does not emphasize the obsolete -ms macros, which have minimal support in System III for the sake of certain manual documents.

Both systems have Versatec drivers expecting a single interrupt address, whereas the Versatec itself has two configured into the hardware. 4.1 at least has comments in the code to tell you this (and #ifdefs to deal with it).

The 4.1 Versatec user programs further expect a Versatec wide enough to handle four pages abreast. If you have an eleven inch unit, as many people do, you must change about half a dozen programs to account for this.

Berkeley has an extensive font library, which System III lacks. These fonts are used by the Versatec filters to simulate the mounted fonts of a C/A/T phototypesetter, the standard destination device for non-device independent *troff*.

The best version of *troff* comes with neither of these systems. This is the Device Independent Troff (DITroff) package, available separately from Bell, and includes useful graphics packages (*pic* and *ideal*) which can be used to augment the basic typesetting facilities.

### 3.4 Graphics

4.1 has rather rudimentary graphics capability.

In contrast, System III has the PWB graphics package, including *ged*, a graphical editor, and numerous data generation, transformation, and display

commands. This graphics capability has been used extensively in the accounting display packages.

### 3.5 SCCS

System III includes the PWB Source Code Control System (SCCS), not available in 4.1bsd. 4.2 is rumored to include "RCS", a public-domain rendering of SCCS.

### 3.6 Ingres

According to "Installing and Operating 4.1bsd," An additional 1200' tape supplied with the distribution contains... the INGRES data base system...." While we found most of the things that were supposed to be on this tape (such as the font library), INGRES was not among them. This was apparently an oversight, as INGRES is a standard part of the distribution and does appear on the great majority of the distribution tapes.

System III has no data base facility.

### 3.7 Text Editors

The only editor common to the two systems is the traditional UNIX editor, *ed*. 4.1 also has the *vi* screen editor and the associated line editor *ex* (an elaboration of *ed*). Recent versions of the Rand Editor *e* and UNIX Emacs can presumably be made to run correctly on System III, although this was not our experience. No problems were noted running them under 4.1.

### 3.8 Mail

System III has only a rudimentary mail system, not much altered from V7.

4.1 has a more elaborate one, with most of the commonly useful mail functions. 4.1 actually has two mail delivery routes, one unprotected and the other encrypted.

## 4. System Calls

Most of the system calls are the same. Several differences are worth mentioning, however.

### 4.1 Vfork

Berkeley UNIX includes the *vfork* version of the *fork* system call, to allow creation of a new process without the need for copying the entire address space of the parent. This makes sense in the paging environment provided by 4.1 (see comments below), but the implementation also imposes certain restrictions which can mislead the unwary. Performance statistics relating to the use of *vfork* are widely available, and are outside the domain of this presentation.

## 4.2 *Reboot*

4.1 has the *reboot* system call, which is quite convenient for persons engaged in system development work. (See below on the reboot command.)

## 4.3 *Setpgrp*

4.1 has elaborated the *setpgrp* system call to be more compatible with the job control functions of the Cshell.

## 4.4 *Ioctl*

The actual *ioctl* system call is identical in the two systems. The interesting differences are in the terminal driver *ioctls*. Both drivers utilize the "line discipline" notion, allowing dynamic choice among several protocols by the user process.

Berkeley added several new features to the V7 terminal driver in 4.1bsd. Some of these are accessed as a new line discipline (the "new tty" discipline), while a few others are implemented as additional *ioctl* calls. (4.1 also includes the BERKNET line discipline.) All of these are useful features, but the tty *ioctls* have become somewhat baroque.

The System III terminal driver is radically different from the V7 one. Many functions which should always have been orthogonal are. As one example, conversion of carriage return to new line on input, and of new line to carriage return and line feed on output, are now separately controllable functions. Of course, this driver is incompatible with any previous one (and with the Berkeley one), but is said to be preserved by System V. There is peripheral processor support for this line discipline in the KMC-11, also (see below).

## 4.5 *Fcntl* and *Open*

The *dup2* function of V7 and 4.1bsd has been replaced and elaborated on in System III by the *fcntl* system call. *Fcntl* also allows control over which file descriptors are inherited across an *exec*. In conjunction with an additional argument (mode) to the *open* system call, *fcntl* permits access to the O\_NDELAY (non-blocking I/O) capability.

We note that there is a bug which must be corrected if O\_NDELAY is to work properly.

In *tty.c/canon*, it is necessary to add a test for FNDELAY:

```
while (tp->t_delct == 0) {
/* jsq 16 September 1982 add FNDELAY per sch@mitre-bedford */
    if ((tp->t_state & CARR_ON) == 0 || u.u_fmode & FNDELAY) {
        spl0();
        return;
    }
    tp->t_state |= IASLP;
```

In *tty.c/ttread*, the check for FNDELAY needs to be removed:



```
    if (tq->c_cc == 0)
/* jsq 3 July 1982 z@cca's fix for NDELAY, changed from:
    if (u.u_fmode&FNDELAY)
*/
/*
* jsq 16 September 1982 sch@mitre-bedford's fix, changed from:
    if (u.u_fmode&FNDELAY && tq->t_rawq.c_cc == 0)
        return;
    else
*/
        canon(tp);
```

Note there are two successive fixes here, the first of which almost worked.

#### 4.6 *Plexes vs. FIFOs*

4.1 utilizes multiplex files for interprocess communication. An alternative but not quite corresponding feature in System III is the named pipe, or FIFO. We have seen relatively few programs actually use either of these features, and note that plexes are supposedly being phased out of both Bell and Berkeley UNIX in favor of more generally useful concepts, particularly the `socket` IPC of 4.2bsd.

### 5. *Libraries and Subroutines*

There are many changes in this section.

#### 5.1 *Software Signals*

4.1 has added additional software signals to support the job control features of the Cshell; these are related to the new tty line discipline.

System III has a so-called software signal facility, *ssignal*(3C), intended to allow exception handling on error conditions. It reportedly works but is not widely used; we in fact have no personal experience with the package. It is also rumored that System V math library errors are trapped to a handler which may be replaced by the user, a useful feature if true.

#### 5.2 *Stdio Buffering*

System III stdio buffers output to everything but terminals, which are handled a character at a time. 4.1 buffers output even if the output file is a terminal, but flushes all terminal or pipe output when the process attempts to read from a terminal or a pipe.

Both systems keep `stderr` unbuffered.

#### 5.3 *Printf*

System III has changed the internal calling sequence of *printf* and eliminated the "%r" format in favor of the (undocumented) *vprintf* and *vfprintf* subroutines. This means any code which depends on "%r" or directly calls `_doprnt` (which no longer exists) will fail.

In a similar vein, certain Berkeley programs assume that *sprintf* returns the address of the buffer, an undocumented feature that has been changed and properly documented in System III.

System III *printf* has abolished the old capital letter formats ("%X", "%F") for long variables in favor of the prepended-l' ("%lx", "%lf") format, so that capital letters can be used in the hexadecimal and floating point formats to mean capital letters in the output stream.

4.1 has basically the V7 *printf* and *scanf*.

The *printf* and *scanf* formats are incompatible in both systems, though we are given to understand that this is no longer quite so true on System V.

#### 5.4 String routines

On System III, the V7 (and 4.1) *index* and *rindex* functions have become *strchr* and *strrchr*, respectively. There are also a few additional string routines reminiscent of certain SNOBOL pattern primitives.

#### 6. Devices

Details of device drivers are beyond the scope of this paper. We only mention a few corresponding to the most important devices.

##### 6.1 Tty

See above under *ioctl* for a discussion of the terminal driver changes.

##### 6.2 KMC-11

System III uses the DEC KMC-11 peripheral processor for several functions, such as offloading most terminal I/O processing, and implementing the Virtual Protocol Machine (VPM) functions.

Operating under System III, we saw substantial improvement in throughput with the KMC-11 support for DZ's. Most terminal interrupts and most (input) line discipline functions are handled by the KMC-11. Some problems exist: for example, the KMC-11 code does not support XCASE mode (especially useful for Braille terminals) at all. 4.1 does not support this function properly either.

##### 6.3 VPM

The Virtual Protocol Machine (VPM) is a package which supports a high-level protocol-definition language to be handled by an interpreter running in a KMC-11.

#### 7. File Formats

A few file formats are worth brief mention.

### 7.1 *a.out*

The details of the binary object file format for commands are sufficiently different between the two systems that it is not possible to run an object file from one system on the other.

### 7.2 *ar*

The *ar* archive magic numbers are different between the two systems. 4.1 has the *ranlib* program for inserting an index at the beginning of an archive, so that the archive can be randomly accessed.

### 7.3 *fs*

4.1 uses 1024 byte blocks in its file system and kernel buffers for efficiency, while System III uses the old 512 byte size. It is possible to have two super blocks kept in one in-core kernel buffer. There have been some changes for robustness in the file system mechanism.

We hear System V has adopted these Berkeley file system modifications.

4.2bsd has an extended file system which includes very long file names, and symbolic links. There are also performance modifications, which, experiments at Berkeley indicate, can improve performance by up to a factor of ten.

## 8. *Games*

Both systems provide a variety of games, ranging from the ever-popular hunt-the-wumpus to chess and automated Dungeons and Dragons.

### 8.1 *Broken System III Games*

Most of the System III games produce this message on a VAX:

this game does not work on the VAX

### 8.2 *4.1bsd ASCII Graphics Games*

4.1 has numerous games which use termcap and curses to produce ASCII graphics on various terminals. Examples are *rogue* (a role-playing game in the manner of Dungeons and Dragons), *worms*, *rain*, *canfield*, and *mille*.

### 8.3 *PDP-11 Compatibility*

4.1 provides a package which allows the use of the PDP-11 compatibility feature of the VAX processor. This package is, for some reason, hidden under games.

## 9. *Miscellany*

A few subjects and facilities worth mentioning do not fit well into the manual section categories.

## 9.1 Accounting

System III provides extensive system accounting software. Much of it uses the graphics facilities to automatically produce charts showing various system parameters (disk reads and writes per head, number of swaps in and out, kernel buffer statistics, etc.). These have useful impact in justifying your facility to upper-level management.

4.1 has kernel hooks to collect similar accounting information (including paging statistics), but lacks the graphical output facilities. The facilities provided proved quite adequate for the purposes of actual system management in a non-billing environment, however.

## 9.2 Termcap, curses, etc.

4.1 includes the *termcap* facility, which maps common terminal control functions to the specific escape sequences for a particular terminal, and the *curses* library of cursor motion optimization functions. These are used by a number of programs, including the *vi* editor, to achieve a reasonable degree of terminal independence.

## 9.3 Sources

System III has a number of apparently gratuitous changes to the names of source directories and files (the kernel `"sys"` subdirectory becomes `"os"`, and `"dev"` becomes `"io"`). There is an appropriate makefile for just about everything, however, so that it is possible to go to the appropriate parent directory for a software package and let *make* do the work.

The 4.1 sources are mostly arranged along traditional V7 and 32V lines. (The 4.2 kernel sources are radically reorganized, but along functional lines, for understandable reasons.)

There is a rather widespread problem in Berkeley code consisting of the use of the type `int` when `long`, or even `off_t`, or especially `time_t` is meant. This works fine, as long as you never try to run such code on a machine where `int` is smaller than 32 bits. (This problem is not evident in the kernel, but rather, in application programs.)

## 10. Maintenance

System III is intended to be a robust production system, while 4.1 is an experimental system. Our experience has been, paradoxically, that 4.1 is more robust than System III.

### 10.1 Shutdown, Reboot, Halt

4.1 has the convenient commands *shutdown* (bring the system down politely, informing the users), *halt* (stop the system immediately), and *reboot* (shutdown and bring up a new system). When coming up, 4.1 automatically performs *fsck* on all the file systems (running one *fsck* subprocess per disk arm, in parallel, for speed) and brings the system up in multi-user mode. To bring 4.1 up from a dead

start, it is only necessary to turn the power switch on. (To get into single user mode, one usually types a C.)

System III has a *shutdown* command (actually a Bourne shell script) as well, but the distributed version did not work very well.

The System III *init* has changed to a finite state machine with up to nine states (1 through 9). The control information is kept in the file */etc/inittab*. States 1 and 2 correspond by convention to the traditional single- and multi- user modes. The system always comes up in state 1. There are no automatic *fsck* checks, though there is provision for auto-restart after power fail.

The terminal information kept in */etc/ttys* on other systems has been combined with */etc/inittab*. If you modify */etc/inittab* to disable lines and then inform *init* of it (by doing "init 2" as superuser), *init* does not disable the lines, though it will notice additions.

## 10.2 Backups

4.1 uses *dump* for file system backups, in the V7 manner.

System III recommends using *volcopy*, a program that does binary disk image copies, with provisions for checking file system and disk volume labels.

## 10.3 Fsck, fsdb

System III has, in addition to *fsck*, a file system debugger called *fsdb* that we never had occasion to use.

4.1 has added the *-p* option to *fsck*, which allows concurrent checking of file systems on different disk arms to speed rebooting.

## 11. Which System to Use

We suspect the main reasons for choosing one of these two systems will have to do with performance, network support by the operating system, or vendor support for the operating system.

### 11.1 Performance

This is a sticky issue which we will not treat in detail, as this is not a performance evaluation presentation. It is useful to mention two qualitative performance areas, however.

#### 11.1.1 Paging vs. Swapping

System III swaps, 4.1 pages. With enough memory on a VAX-11/780, it is difficult to tell the difference for a load of small processes, because System III just doesn't swap. If it is desirable to run huge graphics processes or many Emacs editors or the like, the telling point is not so much the performance as the virtual address space provided by the 4.1 paging system.

We certainly do not intend to indicate, however, that we think paging and swapping produce equivalent performance. There are many technical papers on comparative performance that indicate paging gives much better performance; it is merely that our (admittedly idiosyncratic) experience was that under a light load it is hard to tell the difference without measuring it.

#### 11.1.2 *Terminal I/O*

Using DH11 terminal controllers, 4.1 provides reasonable terminal I/O performance. Berkeley has modified the DZ11 driver sufficiently that even these (basically interrupt per character) devices are usable. It should be remembered that DEC does not provide DH11 controllers for VAXen. This affects DEC maintenance, though similar hardware is available from other vendors.

If you need numerous terminals running at 9600 baud or higher, the System III combination of DZ11s and KMC11 terminal controllers seems far preferable.

#### 11.2 *Network Support*

Both systems have uucp, each version elaborated in different directions. In addition, 4.1 has the USENET news facility.

There is also a package from BBN which provides ARPANET support for 4.1.

Those who want true network support must wait for 4.2.

It is rumored that System V will have TCP/IP support of some kind, and will be available about the same time as 4.2bsd. This claim is disputed by some; we have no evidence one way or the other.

#### 11.3 *Vendor Support*

Many companies bringing out new Motorola 68000-based systems recently have chosen System III as the base for their operating system. To some extent, this will no doubt lock them into System III, probably to be followed by System V. Thus, those wanting to buy something close to a small turnkey system will probably wind up with essentially Bell UNIX.

We note, of course, that there is at least one well-known vendor who will be supporting 4.2bsd on the 68000. Others will probably follow.

There is a large amount of free software available for 4.1bsd that was written principally at academic institutions. Much of it is portable to System III; some of it is of questionable quality.

Most commercial vendors attempt to produce and sell software packages to run on either variety of UNIX. Bell is among these vendors, with the DITroff package, the "S" statistical package, etc.

Many of the commercial vendors using System III have produced graphical, menu-driven interfaces for the naive user, so that it is never necessary to deal directly with any UNIX shell. These mostly require bit-map terminals, varieties of which are also available from other vendors.

#### 11.4 *Combinations*

For companies with the resources, the best solution is probably to run either System III or 4.1bsd and port the desired facilities of the other. This is the traditional route.

Even companies with no desire to merge the two systems would be well-advised to get some sort of expert support (whether in-house or not), as neither Bell nor Berkeley can be counted on to offer the really broad support traditionally supplied by hardware vendors for their operating systems.

#### 11.5 *Merger*

In recent years, Berkeley and Bell have followed increasingly divergent paths. The extended file system and network additions of 4.2bsd are at least as useful as the rational terminal control and Virtual Protocol Machine of System III (and V), but it is not possible to get all of these things in the same operating system as distributed by any one source.

Unfortunately, it is not possible for Berkeley to include software from Bell licences later than 32V, because the price would be prohibitive for many of the Berkeley licensees (there are academic licenses for System III, but they are much more expensive than those for 32V). It is said that System V includes a number of 4.1 facilities already (such as *vi* and *termcap*). If so, this is a step in the right direction.

The real solution to the question of which UNIX to use would be for Bell and Berkeley to make an attempt to merge the two divergent Unix lineages into one operating system again. The practicality of such a merger at this late stage is debatable, of course.

---

# Contiguous Load Modules for UNIX

*Steve Zucker*

Interactive Systems Corporation  
Santa Monica, CA 90401

Most UNIX file input/output is performed a block at a time, the only exceptions being swapping and "raw" disk I/O for which access to an entire logical disk is required. The normal handling of disk allocation and access is quite reasonable for most uses, especially on a multiuser, multiprocess timesharing system in which the independent execution streams tend to randomize accesses to the disk, and in which programs request I/O in block-sized units. However, the loading of programs in response to the *exec* system call is an obvious candidate for multi-block, contiguous transfers. This is all the more important since program loading is critical to user-perceived response time; there is "dead time" from the issuing of a command until the requested program is loaded. While any UNIX system would benefit from contiguous load modules, they are practically a necessity for systems based on slow-access, high-rotational-latency disks such as today's floppies.

Two components are required to support contiguous load modules. First, there must be a utility to reorganizes a disk so that the blocks of load modules are stored contiguously. Second, the operating system must be modified to recognize that a program to be loaded is a contiguous file and to process the load module accordingly. A relatively simple reorganization of the dump and restor utilities accomplishes the disk reorganization. Because the memory into which a program is to be loaded is already locked down in *exec*, it is also a relatively simple matter to have the system read a contiguous load module in a single raw disk operation, once it recognizes that the load module is contiguous.

There are three ways that contiguous load module might be recognized: an indication in the file header, an indication in the file mode word, and dynamic recognition. An indicator in the file header would be "forgeable" and has serious security implications. Changing the file mode word or other part of the inode would introduce a serious incompatibility with existing programs. Dynamic recognition, in which the information is gleaned from the block pointers in the inode and indirect blocks themselves is certainly the method of choice. The number of additional disk accesses for reading indirect blocks is on the order of one percent of the number of blocks in the file. Furthermore, files that are not contiguous would normally show up as such within the inode itself, so the overhead involved in the check is negligible for non-contiguous files. In addition, it would probably be worthwhile to take advantage of files that were "almost" contiguous or even those that had only a few contiguous blocks by scheduling input of in-sequence blocks as multi-block transfers, but allowing out-of-sequence blocks by breaking up the input appropriately.

## Improved Schedulers for Non-Paged UNIX Systems

*Darwyn Peachey*

Hospital Systems Study Group  
Saskatoon, Sask. CANADA S7H 4K1

The design of the process scheduler used in a timesharing system can greatly affect the responsiveness of the system, particularly in the case of a non-paged computer under heavy load. At HSSG we have replaced the "vanilla" UNIX Version 7 scheduling algorithms with a new scheduler based on the FB(n) feedback queue scheme. The new scheduler also features: (1) a greater degree of communication between the CPU scheduling and swap scheduling routines; and (2) an improved approach to the management of main memory, aimed at reducing the space lost to external fragmentation.

The presentation will explain the algorithms used in the modified scheduler. The performance effects of the modified scheduler will be illustrated using benchmark results from various PDP 11's with UNIX Version 7 and System III.



---

# An Implementation of the *vfork* System Call for PDP-11 UNIX

*Michael Karels*

University of California, Berkeley  
Berkeley, CA 94720

A *vfork* system call has been implemented for the Version 7 PDP-11 UNIX system. This system call duplicates the function of *fork* except that the parent and child processes share data and stack segments until the child process calls either *exec* or *exit*. As most forks are followed nearly immediately by an *exec*, this avoids copying data which will soon be released without being modified. This method of process creation thus saves the cpu time or swap needed to generate a new copy. The implementation uses a form of scatter loading of processes so that a process in *vfork* can be treated like any other process. Scatter loading has the additional advantage of increasing memory utilization by reducing fragmentation.

Processes are loaded into memory in four separate sections, one each for shared text, data, user stack, and the kernel per-process data and stack. At the time of a *vfork*, a new proc entry and user structure are allocated and initialized in the same way as for *fork*. However, instead of copying the data and stack for the child, the parent's memory is temporarily associated with the child process. The parent then waits for the child to finish with its data. When the child completes the *vfork* with an *exec* or *exit*, it notifies the parent that it can reclaim its resources and continue execution. The majority of the changes to the kernel are related to scatter loading; the changes for *vfork* itself add about 45 lines of code.

The *vfork* system call allows creation of a new process in a much more efficient manner than with *fork*, saving 60% of the overhead of a *fork*. It was originally implemented on the VAX because of the problems associated with the copy in *fork*. This modification is most useful for the shell, *csh*, and the editor, *ex*, as well as any other programs with large data segments that must *fork* in order to start subprocesses. Scatter loading and *vfork* for the PDP-11 are available as a part of the Second Berkeley Software Distribution (2BSD).

---

## Handling Very Large Programs on a 16-Bit Super-micro

*Mitch Bishop*

Zilog, Inc.  
Campbell, CA 95008

Many of the "super-micros" in today's marketplace are offering more functionality at a continually lower price. As system hardware becomes more powerful and cheaper to produce, system software takes on the burden of making a computer marketable and usable. Zilog's entry in the 16 bit "super-micro" market is the System 8000, a powerful multi-user system that backs up excellent performance with system software called ZEUS (Zilog Extended UNIX System).

The System 8000 was first shipped in August of 1981 with a full UNIX Version 7 port, along with many new hardware and software features. Its initial performance was comparable to that of the PDP-11 series running UNIX. (See *Mini Micro Systems*, "Challenging the Minis" September 1981.)

The ZEUS Operating System was updated in June 1982 with further extensions to support the execution of very large (greater than 128kb) programs. Utilizing the efficient segmented architecture of the Z8001 microprocessor, ZEUS offers dramatic advantages over competing UNIX systems.

---

## Handling Very Large Programs on a 16 Bit "Super-micro"

Mitch Bishop  
Operating Systems Project Manager  
Zilog, Inc.  
1315 Dell Ave. Bl-2  
Campbell, CA 95008

(408) 370-8000 ext. 4522 between 8 a.m. and 6 p.m.

### INTRODUCTION

Many of the "super-micros" in today's marketplace are offering more functionality at a continually lower price. As system hardware becomes more powerful and cheaper to produce, system software takes on the burden of making a computer marketable and usable. Zilog's entry in the 16 bit "super-micro" market is the System 8000, a powerful multi-user system that backs up excellent performance with system software called ZEUS (Zilog Extended UNIX\* System).

The System 8000 was first shipped in August of 1981 with a full UNIX Version 7 port, along with many new hardware and software features. Its initial performance was comparable to that of the PDP-11 series running UNIX. (See Mini Micro Systems, "Challenging the Minis", September 1981).

The ZEUS Operating System was updated in June, 1982 with further extensions to support the execution of very large (greater than 128K byte) programs. Utilizing the efficient segmented architecture of the Z8001 microprocessor, ZEUS offers dramatic advantages over competing UNIX systems.

### HARDWARE OVERVIEW

The Z8001 microprocessor has two modes of operation: non-segmented and segmented. Non-segmented mode is a 16 bit architecture that allows addressing of four spaces of 64k each: system code and data; user code and data. Processes executing on the Z8001 in non-segmented mode therefore have a maximum memory space of 128 kbytes.

The Z8001, in segmented mode, combines the 16 bit offset with a seven bit number, called the segment number, to address a memory space of 8 Mbytes. There is a special Z8001 instruction that allows a system to dynamically switch

---

\*UNIX is a Trademark of Bell Laboratories.

---

between non-segmented and segmented mode.

Address translation is handled by Z8010 Memory Management Units. There are three MMUs in the System 8000, for code, data, and stack translations. Each MMU has 64 segment descriptors. Each segment descriptor can address a segment of physical memory that can be as little as 256 bytes or as large as 64K bytes. The 23 bit logical address generated by the Z8001 is converted into a 24 bit physical address by the MMUs.

The Z8001 generates the 7 bit segment number one cycle ahead of the rest of the address, which means that the MMUs are set up for translation by the time the offset comes along. This avoids fetching and translating the two pieces of the address separately.

#### ADVANTAGES OF SEGMENTATION

The major advantage of segmented addressing is the ability to dynamically relocate code anywhere in memory without affecting logical addresses contained in the code. This feature allows efficient swapping of user programs in and out of memory, since the ZEUS kernel does not guarantee a user program the same physical load address the code previously occupied.

Segmentation also offers a sophisticated protection mechanism. On every address translation, an MMU will compare the attributes of the physical address with that of the instruction using that address. If the attributes do not match, then an exception interrupt is generated. With this scheme, the ZEUS kernel address space is protected from user program access. This feature also allows automatic stack growth for user programs that run out of their allocated space.

#### SEGMENTED INSTRUCTION SET

The instruction set used to program the microprocessor in segmented mode is symmetrical to the non-segmented instruction set. The only differences are instructions that must be slightly altered to include the 7-bit segment number. For example, instructions that deal with addresses use registers in non-segmented mode and register pairs in segmented mode. In most instances the instruction format is exactly the same and the compiler is able to generate segmented code with few changes.

#### CREATING SEGMENTED PROGRAMS UNDER ZEUS

A segmented version of the portable C compiler is used to generate segmented instructions that can run through the ZEUS assembler. In addition, a new loader called sld(1) is

---

used to link and load segmented code. The user can specify that certain segments are to be used for code or data, or the loader can just use default segment assignments. In the simplest case, the procedure for creating segmented programs is exactly the same as for the non-segmented case.

### ZEUS KERNEL IMPLEMENTATION

The release of segmentation support in no way affects the operation of non-segmented processes. Non-segmented user programs need not be re-compiled to run with the new support kernel. The basic philosophy in making the changes to the ZEUS kernel to support segmented users was to treat the non-segmented case as a sub-set of the segmented case. Wherever possible, kernel code is shared between the two cases. In some cases (i.e. `exec(2)`), the code for the segmented case was kept completely separate for performance reasons.

The task of supporting segmented programs in the kernel was accomplished in three steps. The first and most obvious step involved changing all 16 bit user address references to 32 bits. The next major piece of work involved re-writing all the code that deals with the MMUs. The accompanying figures show how the ZEUS kernel uses the three Z8010 MMUs to address code, data, and stack for non-segmented and segmented users. The last major task was to create the code to start a segmented program and supply the special system calls to dynamically alter the configuration of a segmented program.

Two assumptions were made in converting the kernel to run segmented user programs: 1. processes reside either entirely in memory or are entirely swapped out to disk; and 2. a single process' address space is contiguous. These assumptions make the task of swapping and memory allocation straightforward. Upon swapping into memory, the MMUs are reset to map the new physical locations with the same segment boundaries and protections.

The ZEUS operating system allows a segmented user program to occupy up to 122 segments for code or data. Six segment descriptors are reserved for the kernel. This gives complete flexibility to the user in the number and size of the segments in a particular user program.

### NEW SYSTEM CALLS

The user can allocate segments to a program in two ways: through the segmented loader scheme described earlier; or at run time through the `mkseg(2)` system call. The program can specify a preferred segment number and an initial size to the newly created segment. If the requested segment number is already in use, then the call returns an error.

---

Otherwise the call returns a pointer to the start of the new segment.

There are two ways a segmented program can manipulate the size of its data area; the `sbrk(2)` and `ssbrk(2)` system calls. These calls are analogous to the non-segmented versions `brk(2)` and `sbrk(2)`. The `sbrk(2)` call takes a segmented address as an argument and sets the highest data address to this value. The `ssbrk(2)` call takes a segment number and increment as arguments and changes the size of the given data segment. In both cases a pointer to the start of the new data space is returned.

#### PERFORMANCE CONSIDERATIONS

For the most part, the differences between segmented and non-segmented code remain completely transparent to the programmer and user of an application program.

Segmentation does introduce some factors that users should be aware of when deciding to code a program non-segmented or segmented. Segmented programs require larger addresses, as mentioned before, to include the segment number. This results in segmented versions of programs being 10 to 20 percent larger than their non-segmented counterparts. Also, non-segmented instructions are executed slightly faster than segmented instructions when address fetches occur. This translates to segmented programs executing 10 to 15 percent slower than the non-segmented versions. The user should evaluate whether the application requires more than 64k code or 64k data. If so, then segmented code is appropriate. Otherwise, the user should gain the faster execution time and smaller memory requirements by coding in non-segmented mode.

Additionally, the programmer may not declare any single data item to be larger than 64K. Data may of course greatly exceed 64k, but any single variable must have a size of less than 64k. This does not pose a serious limitation.

However, in relation to other UNIX processors, the Z8001 architecture is much superior because of its 16 bit (non-segmented) mode of operation. Other machines, based on the 68000 chip, for instance force user programs to carry 32 bit addresses around, even when the programs are not very large. PDP-11 machines solve the addressing problem by not supporting programs larger than 128K! Clearly the implementation of segmentation takes advantage of the two modes of Z8001 operation, while retaining the high performance of a 16 bit architecture.

---

## EXAMPLES

The largest segmented program that we have written is a C global optimizer, which has a minimum data space of 170K bytes. The global optimizer uses 9 dynamically allocated data segments to manage separately growing data spaces. These spaces are used for flow graphs, symbol tables, and the generation of intermediate code.

While testing the optimizer, we discovered that the overhead of the `mkseg(2)` and `ssgbrk(2)` calls was much more than expected. This was caused by the fact that the kernel was copying the entire segmented programs address space from one place in memory to another. With some testing, we eliminated the problem by increasing the size of the data increments being requested.

Another interesting segmented program is a version of the standard UNIX dump utility. While trying to keep our nine-track tape drives in a streaming mode, we discovered that our disks were too slow to keep up with the tape drive. By creating a 64K data buffer for data transfers, however, a segmented version of dump could run fast enough to keep the nine-track drive streaming.

These examples point out the fact that segmented user programs on the System 8000 have a powerful new way to manage data areas that are virtually unheard of in similar sized machines.

## SUMMARY

The ZEUS segmented support tools offer a dramatic functionality advantage over many competitive UNIX systems:

ZEUS offers a set of simple tools that allow the user to take advantage of segmentation in much the same way as 16-bit applications are programmed.

ZEUS offers the possibility of vastly larger processes than can be supported by pure 16 bit systems such as members of the DEC PDP-11 family or the Zilog Z8002.

It offers improved structure for shared memory versus machines with a linear address space. In particular, shared segments are less sensitive to the actual addresses used in the sharing processes and can be more readily protected.

It allows for innovative software architectures that use the features to dramatically improve performance.

---

It is completely compatible with the existing software base.

The ZEUS segmented support tools give the user a choice when writing new applications. Non-segmented programs are optimized for speed and size while segmented programs can be used for very large



Chairperson: *Larry K. Isley*  
AT&T

## System V Offering

*W. R. Guffy*

AT&T

The UNIX kernel has evolved to stability with System V. There have been significant performance enhancements from within and outside Bell. The current functionality is “unalterable” although there could be additions in future releases. There will be future releases of UNIX beyond System V to improve performance and/or add new features. AT&T is committed to upward compatibility with future releases beyond System V. These releases may well be unbundled since the basic system and kernel are to be unalterable.

Mr. Guffy made the point that offering supported software is a new venture for Western Electric. He stated that they are not intending to displace existing companies that are offering products and support.

The documentation on System V is better than that for System III: there is more and there are more types of documents. Licensing contacts for System V remain the same except for overseas which will be handled by AT&T International.

Training and technical seminars on System V will be offered; availability will be announced in the first quarter of 1983.

## System V Support Offering

*Dave Sandel*

Western Electric

System V will be supported by Western Electric from  
UNIX Systems Support Center  
2600 Warrenville Road  
Lisle, IL 60532  
(312) 260-4880

Three different markets to be supported were named — domestic, international, and educational — although it was not stated what different treatment these markets might receive.

The basic System V product runs on the VAX 11/780 or 11/750 or the PDP 11/70. A complete list of the supported configurations can be obtained from AT&T Technology Licensing.

There are two levels of support offered. Level 1 includes:

- a monthly newsletter with product-related information and offerings, in-depth discussion of the most recent major problem uncovered, and support center activity and procedure updates;

- periodic updates with bug fixes (only);
- problem reporting capabilities including a guest login; and
- a periodically-updated known problem list with all known System V problems.

Level 2 includes all level 1 services plus more immediate support through a hotline 800 phone number.

One copy of each of the 12 System V documents comes with System V. Additional copies of each may be purchased. It appears from the list of documents that there is much more, and more useful, documentation available than with previous versions of UNIX.

The support center also offers an electronic mail facility for its customers. It is used with an automatic calling unit and *uucp*.

## Licensing Activity and Pricing

*Larry K. Isley*

AT&T  
Greensboro, NC 27420

Five C compilers and three versions of phototypesetter software are being offered for vendors to offer to their customers. The end-user offering must be binary code only. The compilers are C/370, C/SEL, C/6000, C/DATA, and C/CRAY. The phototypesetter versions are typesetter-independent troff, phototypesetter/PWB, and phototypesetter/V7. Any C compiler or phototypesetter licensee may also provide their customers the C compiler or phototypesetter software from any of the UNIX systems. No up-front fee is required. The end-user fee for any of these is \$200.

The following table shows the number of UNIX source licenses and installations worldwide as of December 1, 1982.

Software	Commercial		Educational & Admin.		Government		Total	
	lic	inst	lic	inst	lic	inst	lic	inst
Mini	6	8	128	376	0	0	134	384
V6	90	170	360	958	63	180	513	1308
PWB	48	133	65	246	86	141	199	520
V7	144	237	359	1092	101	175	604	1504
32V	71	121	206	435	36	59	313	615
SIH	176	265	4	4	15	15	195	284
<b>TOTALS</b>	<b>535</b>	<b>934</b>	<b>1122</b>	<b>3111</b>	<b>301</b>	<b>570</b>	<b>1958</b>	<b>4615</b>

Commercial source licensing fees are given in the following table along with the fee to upgrade to System V if you have the specified license(s).

Software				Upgrade to System V	
	Initial CPU	Additional CPU	Customer CPU's	Initial CPU	Additional CPU
Mini	12,000	4,000			
V6	20,000	6,700	8,400	26,000	10,300
PWB	30,000	10,000	12,000	16,000	7,000
V7	28,000	9,400	11,700*	18,000	7,600
32V	40,000	15,000	18,000	6,000	2,000
SIII	43,000	16,000	*	1,000**	
SV	43,000	16,000	*		
UNIVAC	30,000	10,000	12,000		
TSS	100,000	20,000			
V6+V7				14,000	6,300
V7+PWB				4,000	3,000

\* other schedules apply

\*\* one time for all CPU's

Educational and administrative licensing fees to upgrade to System V are given in this table.

	Ed.	Admin.
Initial request covering all CPU's	800	16,000
Each additional CPU after initial request	400	400
Upgrade all CPU's from System III	0	0
Upgrade all CPU's from other UNIX license	800	16,000*

\* some credit allowed for previous administrative licenses

The fees for System V support for a data center are:

level 1 \$150 per month for the first CPU and \$50 per month for each additional CPU.

level 2 \$350 per month for the first CPU and \$100 per month for each additional CPU.

A fee of \$100 per hour is charged for out-of-hours service, or support services not covered in the level 1 or level 2 offering.

Further information on licensing and support contracts is available for domestic sites from AT&T Technology licensing at (919) 697-6530 and for foreign sites from AT&T International at (201) 953-7581 or TELEX-219365 ATT-I-UR.

Chairperson: *Greg Noel*  
NCR Corporation

## The Plexus Networked UNIX

*Monte Pickard*

Plexus Computers, Inc.  
Santa Clara, CA 95050

Plexus has designed and implemented a user transparent Distributed File System and Virtual Terminal capability under the standard UNIX System III.

The file system involves an extension of the *mount(1M)* command such that any branch of any file system on the network may be mounted on a branch of the local file system. Accesses to a path that crosses that mounted branch are satisfied on the remote system. All buffering is done on the system that contains the file system, as well as permissions granted based on the *etc/passwd* of that system.

The Virtual Terminal capability consists of an implementation of the 'tty' characteristics over the network. All programs that utilize a tty interface may utilize the Virtual terminal capabilities. Local, as well as remote efficacy of *ioctl* provides expected response at both ends of the virtual circuit established.

The interface to the user is established at the system call level with the specific details of network media or technology transparent to the system code implementing the desired capability. This is done through the definition of a software architecture for the communication function that provides general virtual circuit and datagram capabilities through implemented networks.

The development was done utilizing the ethernet local area network technology. An X.25 implementation, as well as a broad-band implementation are next to be developed.

## CSNET Status Report

*G. Brendan Reilley*

University of Delaware  
Newark, Delaware 19711

The CSNET project is an inter-institutional effort to provide a nationwide network of computer science research departments. It includes both commercial and academic members, and is intended to be self-supporting, running on a pay-by-use basis. The initial funding has been provided by the National Science Foundation, and this seed money has carried the network into production. The initial implementation of CSNET supports mail transport only, though other services are planned for the future. CSNET currently supports a Coordination and Information Center, two mail relay machines, and a Service Host, which provides a nameserver for the network.

CSNET is a logical network rather than a physical network. This means that it uses existing transport mechanisms, rather than providing a new physical backbone. Currently it uses Telenet, Arpanet, and the dial-up telephone network. Approximately fifty member sites are currently being serviced.

---

**This page intentionally left blank**

---

## Mail Systems and Addressing in 4.2bsd

*Eric Allman*

Britton-Lee, Inc.  
Berkeley, CA 94704

Routing mail through a heterogeneous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an *ad hoc* basis. However, this approach has become unmanageable as internets grow.

*Sendmail* acts as a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both old and new format addresses. The new format is "domain-based", a flexible technique that can handle many common situations. *Sendmail* is not intended to perform user interface functions.

*Sendmail* will replace *delivermail* in the Berkeley 4.2 distribution. Several major hosts are now or will soon be running *sendmail*. This change will affect any users that route mail through a *sendmail* gateway. The changes that will be user-visible will be discussed.

---

## Mail Systems and Addressing in 4.2bsd

Eric Allman†

*Britton-Lee, Inc.*  
1919 Addison Street, Suite 105.  
Berkeley, California 94704.

eric@Berkeley.ARPA  
ucbvax!eric

### ABSTRACT

Routing mail through a heterogeneous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an ad hoc basis. However, this approach has become unmanageable as internets grow.

Sendmail acts a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both old and new format addresses. The new format is "domain-based," a flexible technique that can handle many common situations. Sendmail is not intended to perform user interface functions.

Sendmail will replace delivermail in the Berkeley 4.2 distribution. Several major hosts are now or will soon be running sendmail. This change will affect any users that route mail through a sendmail gateway. The changes that will be user visible are emphasized.

The mail system to appear in 4.2bsd will contain a number of changes. Most of these changes are based on the replacement of *delivermail* with a new module called *sendmail*. *Sendmail* implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible configuration. Of key interest to the mail system user will be the changes in the network addressing structure.

In a simple network, each node has an address, and resources can be identified with a host-resource pair; in particular, the mail system can refer to users using a host-username pair. Host names and numbers have to be administered by a central authority, but usernames can be assigned locally to each host.

In an internet, multiple networks with different characteristics and managements must communicate. In particular, the syntax and semantics of resource identification change. Certain special cases can be handled trivially by *ad hoc* techniques, such as providing network names that appear local to hosts on other networks, as with the Ethernet at Xerox PARC. However, the general case is extremely complex. For example, some networks require that the route the message takes be explicitly specified by the sender, simplifying the database update problem since only adjacent hosts must be entered into the system tables, while others use logical addressing, where the sender specifies the location of the recipient but not how to get there. Some networks use a left-associative syntax and others use a right-associative syntax, causing ambiguity in mixed addresses.

Internet standards seek to eliminate these problems. Initially, these proposed expanding the address pairs to address triples, consisting of {network, host, username} triples. Network

---

†A considerable part of this work was done while under the employ of the INGRES Project at the University of

numbers must be universally agreed upon, and hosts can be assigned locally on each network. The user-level presentation was changed to address domains, comprised of a local resource identification and a hierarchical domain specification with a common static root. The domain technique separates the issue of physical versus logical addressing. For example, an address of the form "eric@a.cc.berkeley.arpa" describes the logical organization of the address space (user "eric" on host "a" in the Computer Center at Berkeley) but not the physical networks used (for example, this could go over different networks depending on whether "a" were on an ethernet or a store-and-forward network).

*Sendmail* is intended to help bridge the gap between the totally *ad hoc* world of networks that know nothing of each other and the clean, tightly-coupled world of unique network numbers. It can accept old arbitrary address syntaxes, resolving ambiguities using heuristics specified by the system administrator, as well as domain-based addressing. It helps guide the conversion of message formats between disparate networks. In short, *sendmail* is designed to assist a graceful transition to consistent internetwork addressing schemes.

Section 1 defines some of the terms frequently left fuzzy when working in mail systems. Section 2 discusses the design goals for *sendmail*. In section 3, the new address formats and basic features of *sendmail* are described. Section 4 discusses some of the special problems of the UUCP network. The differences between *sendmail* and *delivermail* are presented in section 5.

**DISCLAIMER:** A number of examples in this paper use names of actual people and organizations. This is not intended to imply a commitment or even an intellectual agreement on the part of these people or organizations. In particular, Bell Telephone Laboratories (BTL), Digital Equipment Corporation (DEC), Lawrence Berkeley Laboratories (LBL), Britton-Lee Incorporated (BLI), and the University of California at Berkeley are not committed to any of these proposals at this time. Much of this paper represents no more than the personal opinions of the author.

## 1. DEFINITIONS

There are four basic concepts that must be clearly distinguished when dealing with mail systems: the user (or the user's agent), the user's identification, the user's address, and the route. These are distinguished primarily by their position independence.

### 1.1. User and Identification

The user is the being (a person or program) that is creating or receiving a message. An *agent* is an entity operating on behalf of the user – such as a secretary who handles my mail, or a program that automatically returns a message such as "I am at the UNICOM conference."

The identification is the tag that goes along with the particular user. This tag is completely independent of location. For example, my identification is the string "Eric Allman," and this identification does not change whether I am located at U.C. Berkeley, at Britton-Lee, or at a scientific institute in Austria.

Since the identification is frequently ambiguous (e.g., there are two "Robert Henry"s at Berkeley) it is common to add other disambiguating information that is not strictly part of the identification (e.g., Robert "Code Generator" Henry versus Robert "System Administrator" Henry).

### 1.2. Address

The address specifies a location. As I move around, my address changes. For example, my address might change from "eric@Berkeley.ARPA" to "eric@bli.UUCP" or

---

California at Berkeley.



"allman@IIASA.Austria" depending on my current affiliation.

However, an address is independent of the location of anyone else. That is, my address remains the same to everyone who might be sending me mail. For example, a person at MIT and a person at USC could both send to "eric@Berkeley.ARPA" and have it arrive to the same mailbox.

Ideally a "white pages" service would be provided to map user identifications into addresses (for example, see [Solomon81]). Currently this is handled by passing around scraps of paper or by calling people on the telephone to find out their address.

### 1.3. Route

While an address specifies *where* to find a mailbox, a route specifies *how* to find the mailbox. Specifically, it specifies a path from sender to receiver. As such, the route is potentially different for every pair of people in the electronic universe.

Normally the route is hidden from the user by the software. However, some networks put the burden of determining the route onto the sender. Although this simplifies the software, it also greatly impairs the usability for most users. The UUCP network is an example of such a network.

## 2. DESIGN GOALS

Design goals for *sendmail*<sup>1</sup> include:

- (1) Compatibility with the existing mail programs, including Bell version 6 mail, Bell version 7 mail, Berkeley Mail [Shoens79], BerkNet mail [Schmidt79], and hopefully UUCP mail [Nowitz78]. ARPANET mail [Crocker82] was also required.
- (2) Reliability, in the sense of guaranteeing that every message is correctly delivered or at least brought to the attention of a human for correct disposal; no message should ever be completely lost. This goal was considered essential because of the emphasis on mail in our environment. It has turned out to be one of the hardest goals to satisfy, especially in the face of the many anomalous message formats produced by various ARPANET sites. For example, certain sites generate improperly formatted addresses, occasionally causing error-message loops. Some hosts use blanks in names, causing problems with mail programs that assume that an address is one word. The semantics of some fields are interpreted slightly differently by different sites. In summary, the obscure features of the ARPANET mail protocol really *are* used and are difficult to support, but must be supported.
- (3) Existing software to do actual delivery should be used whenever possible. This goal derives as much from political and practical considerations as technical.
- (4) Easy expansion to fairly complex environments, including multiple connections to a single network type (such as with multiple UUCP or Ethernets). This goal requires consideration of the contents of an address as well as its syntax in order to determine which gateway to use.
- (5) Configuration information should not be compiled into the code. A single compiled program should be able to run as is at any site (barring such basic changes as the CPU type or the operating system). We have found this seemingly unimportant goal to be critical in real life. Besides the simple problems that occur when any program gets recompiled in a different environment, many sites like to "fiddle" with anything that they will be recompiling anyway.
- (6) *Sendmail* must be able to let various groups maintain their own mailing lists, and let individuals specify their own forwarding, without modifying the system alias file.

---

<sup>1</sup>This section makes no distinction between *delivermail* and *sendmail*.

- (7) Each user should be able to specify which mailer to execute to process mail being delivered for him. This feature allows users who are using specialized mailers that use a different format to build their environment without changing the system, and facilitates specialized functions (such as returning an "I am on vacation" message).
- (8) Network traffic should be minimized by batching addresses to a single host where possible, without assistance from the user.

These goals motivated the architecture illustrated in figure 1. The user interacts with a mail generating and sending program. When the mail is created, the generator calls *sendmail*, which routes the message to the correct mailer(s). Since some of the senders may be network servers and some of the mailers may be network clients, *sendmail* may be used as an internet mail gateway.

### 3. USAGE

#### 3.1. Address Formats

Arguments may be flags or addresses. Flags set various processing options. Following flag arguments, address arguments may be given. Addresses follow the syntax in RFC822 [Crock82] for ARPANET address formats. In brief, the format is:

- (1) Anything in parentheses is thrown away (as a comment).
- (2) Anything in angle brackets (" $<>$ ") is preferred over anything else. This rule implements the ARPANET standard that addresses of the form

user name  $<$ machine-address $>$

will send to the electronic "machine-address" rather than the human "user name."

- (3) Double quotes ( " ) quote phrases; backslashes quote characters. Backslashes are more powerful in that they will cause otherwise equivalent phrases to compare differently - for example, *user* and "*user*" are equivalent, but  $\backslash$ *user* is different from

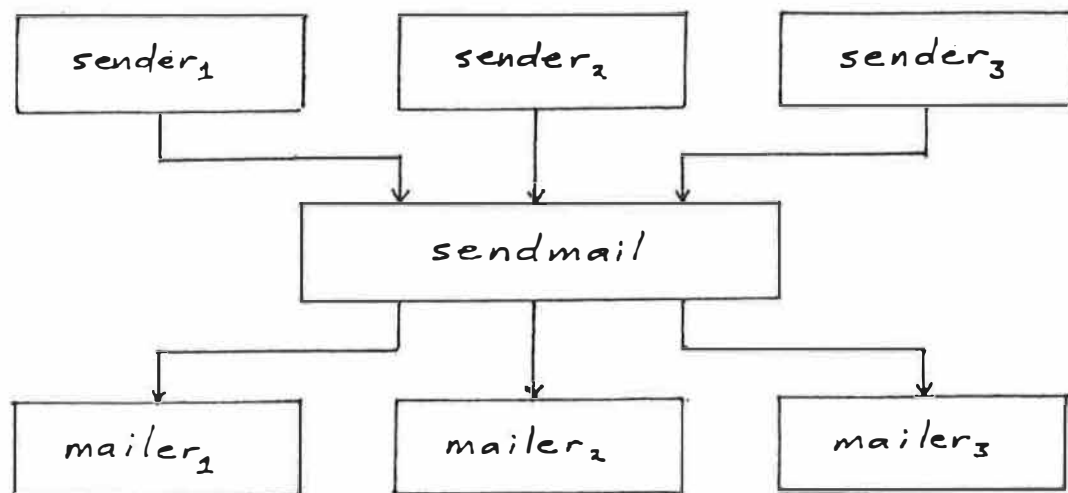


Figure 1 - Sendmail System Structure.

either of them. This might be used to avoid normal aliasing or duplicate suppression algorithms.

Parentheses, angle brackets, and double quotes must be properly balanced and nested. The rewriting rules control remaining parsing<sup>2</sup>.

Although old style addresses are still accepted in most cases, the preferred address format is based on ARPANET-style domain-based addresses [Su82a]. These addresses are based on a hierarchical, logical decomposition of the address space. The addresses are hierarchical in a sense similar to the U.S. postal addresses: the messages may first be routed to the correct state, with no initial consideration of the city or other addressing details. The addresses are logical in that each step in the hierarchy corresponds to a set of "naming authorities" rather than a physical network.

For example, the address:

`eric@HostA.BigSite.ARPA`

would first look up the domain BigSite in the namespace administrated by ARPA. A query could then be sent to BigSite for interpretation of HostA. Eventually the mail would arrive at HostA, which would then do final delivery to user "eric."

### 3.2. Mail to Files and Programs

Files and programs are legitimate message recipients. Files provide archival storage of messages, useful for project administration and history. Programs are useful as recipients in a variety of situations, for example, to maintain a public repository of systems messages (such as the Berkeley *msg* program).

Any address passing through the initial parsing algorithm as a local address (i.e, not appearing to be a valid address for another mailer) is scanned for two special cases. If prefixed by a vertical bar ("|") the rest of the address is processed as a shell command. If the user name begins with a slash mark ("/") the name is used as a file name, instead of a login name.

### 3.3. Aliasing, Forwarding, Inclusion

*Sendmail* reroutes mail three ways. Aliasing applies system wide. Forwarding allows each user to reroute incoming mail destined for that account. Inclusion directs *sendmail* to read a file for a list of addresses, and is normally used in conjunction with aliasing.

#### 3.3.1. Aliasing

Aliasing maps local addresses to address lists using a system-wide file. This file is hashed to speed access. Only addresses that parse as local are allowed as aliases; this guarantees a unique key (since there are no nicknames for the local host).

#### 3.3.2. Forwarding

After aliasing, if an recipient address specifies a local user *sendmail* searches for a ".forward" file in the recipient's home directory. If it exists, the message is *not* sent to that user, but rather to the list of addresses in that file. Often this list will contain only one address, and the feature will be used for network mail forwarding.

Forwarding also permits a user to specify a private incoming mailer. For example, forwarding to:

`"|/usr/local/newmail myname"`

will use a different incoming mailer.

<sup>2</sup>Disclaimer: Some special processing is done after rewriting local names; see below.

### 3.3.3. Inclusion

Inclusion is specified in RFC 733 [Crocker77] syntax:

```
:Include: pathname
```

An address of this form reads the file specified by *pathname* and sends to all users listed in that file.

The intent is *not* to support direct use of this feature, but rather to use this as a subset of aliasing. For example, an alias of the form:

```
project: :include:/usr/project/userlist
```

is a method of letting a project maintain a mailing list without interaction with the system administration, even if the alias file is protected.

It is not necessary to rebuild the index on the alias database when a `:include:` list is changed.

### 3.4. Message Collection

Once all recipient addresses are parsed and verified, the message is collected. The message comes in two parts: a message header and a message body, separated by a blank line. The body is an uninterpreted sequence of text lines.

The header is formatted as a series of lines of the form

```
field-name: field-value
```

Field-value can be split across lines by starting the following lines with a space or a tab. Some header fields have special internal meaning, and have appropriate special processing. Other headers are simply passed through. Some header fields may be added automatically, such as time stamps.

## 4. THE UUCP PROBLEM

Of particular interest is the UUCP network. The explicit routing used in the UUCP environment causes a number of serious problems. First, giving out an address is impossible without knowing the address of your potential correspondent. This is typically handled by specifying the address relative to some "well-known" host (e.g., `ucbvax` or `decvax`). Second, it is often difficult to compute the set of addresses to reply to without some knowledge of the topology of the network. Although it may be easy for a human being to do this under many circumstances, a program does not have equally sophisticated heuristics built in. Third, certain addresses will become painfully and unnecessarily long, as when a message is routed through many hosts in the USENET. And finally, certain "mixed domain" addresses are impossible to parse unambiguously - e.g.,

```
decvax!ucbvax!lbl-h!user@LBL-CSAM
```

might have many possible resolutions, depending on whether the message was first routed to `decvax` or to `LBL-CSAM`.

To solve this problem, the UUCP syntax would have to be changed to use addresses rather than routes. For example, the address "`decvax!ucbvax!eric`" might be expressed as "`eric@ucbvax.UUCP`" (with the hop through `decvax` implied). This address would itself be a domain-based address; for example, an address might be of the form:

```
mark@d.cbosg.btl.UUCP
```

Hosts outside of Bell Telephone Laboratories would then only need to know how to get to a designated BTL relay, and the BTL topology would only be maintained inside Bell.

There are three major problems associated with turning UUCP addresses into something reasonable: defining the namespace, creating and propagating the necessary software, and building and maintaining the database.

#### 4.1. Defining the Namespace

Putting all UUCP hosts into a flat namespace (e.g., "...@host.UUCP") is not practical for a number of reasons. First, with over 1600 sites already, and (with the increasing availability of inexpensive microcomputers and autodialers) several thousand more coming within a few years, the database update problem is simply intractable if the namespace is flat. Second, there are almost certainly name conflicts today. Third, as the number of sites grow the names become ever less mnemonic.

It seems inevitable that there be some sort of naming authority for the set of top level names in the UUCP domain, as unpleasant a possibility as that may seem. It will simply not be possible to have one host resolving all names. It may however be possible to handle this in a fashion similar to that of assigning names of newsgroups in USENET. However, it will be essential to encourage everyone to become subdomains of an existing domain whenever possible – even though this will certainly bruise some egos. For example, if a new host named "bli" were to be added to the UUCP network, it would probably actually be addressed as "d.bli.UUCP" (i.e., as host "d" in the pseudo-domain "bli" rather than as host "bli" in the UUCP domain).

#### 4.2. Creating and Propagating the Software

The software required to implement a consistent namespace is relatively trivial. Two modules are needed, one to handle incoming mail and one to handle outgoing mail.

The incoming module must be prepared to handle either old or new style addresses. New-style addresses can be passed through unchanged. Old style addresses must be turned into new style addresses where possible.

The outgoing module is slightly trickier. It must do a database lookup on the recipient addresses (passed on the command line) to determine what hosts to send the message to. If those hosts do not accept new-style addresses, it must transform all addresses in the header of the message into old style using the database lookup.

Both of these modules are straightforward except for the issue of modifying the header. It seems prudent to choose one format for the message headers. For a number of reasons, Berkeley has elected to use the ARPANET protocols for message formats. However, this protocol is somewhat difficult to parse.

Propagation is somewhat more difficult. There are a large number of hosts connected to UUCP that will want to run completely standard systems (for very good reasons). The strategy is not to convert the entire network – only enough of it to alleviate the problem.

#### 4.3. Building and Maintaining the Database

This is by far the most difficult problem. A prototype for this database already exists, but it is maintained by hand and does not pretend to be complete.

This problem will be reduced considerably if people choose to group their hosts into subdomains. This would require a global update only when a new top level domain joined the network. A message to a host in a subdomain could simply be routed to a known domain gateway for further processing. For example, the address "eric@a.bli.UUCP" might be routed to the "bli" gateway for redistribution; new hosts could be added within BLI without notifying the rest of the world. Of course, other hosts *could* be notified as an efficiency measure.

There may be more than one domain gateway. A domain such as BTL, for instance, might have a dozen gateways to the outside world; a non-BTL site could choose the closest gateway. The only restriction would be that all gateways maintain a consistent view of the domain they represent.

#### 4.4. Logical Structure

Logically, domains are organized into a tree. There need not be a host actually associated with each level in the tree – for example, there will be no host associated with the name “UUCP.” Similarly, an organization might group names together for administrative reasons; for example, the name

CAD.research.BigCorp.UUCP

might not actually have a host representing “research.”

However, it may frequently be convenient to have a host or hosts that “represent” a domain. For example, if a single host exists that represents Berkeley, then mail from outside Berkeley can forward mail to that host for further resolution without knowing Berkeley’s (rather volatile) topology. This is not unlike the operation of the telephone network.

This may also be useful inside certain large domains. For example, at Berkeley it may be presumed that most hosts know about other hosts inside the Berkeley domain. But if they process an address that is unknown, they can pass it “upstairs” for further examination. Thus as new hosts are added only one host (the domain master) *must* be updated immediately; other hosts can be updated as convenient.

Ideally this name resolution process would be performed by a name server (e.g., [Su82b]) to avoid unnecessary copying of the message. However, in a batch network such as UUCP this could result in unnecessary delays.

#### 5. COMPARISON WITH DELIVERMAIL

*Sendmail* is an outgrowth of *delivermail*. The primary differences are:

- (1) Configuration information is not compiled in. This change simplifies many of the problems of moving to other machines. It also allows easy debugging of new mailers.
- (2) Address parsing is more flexible. For example, *delivermail* only supported one gateway to any network, whereas *sendmail* can be sensitive to host names and reroute to different gateways.
- (3) Forwarding and `:include:` features eliminate the requirement that the system alias file be writable by any user (or that an update program be written, or that the system administration make all changes).
- (4) *Sendmail* supports message batching across networks when a message is being sent to multiple recipients.
- (5) A mail queue is provided in *sendmail*. Mail that cannot be delivered immediately but can potentially be delivered later is stored in this queue for a later retry. The queue also provides a buffer against system crashes; after the message has been collected it may be reliably redelivered even if the system crashes during the initial delivery.
- (6) *Sendmail* uses the networking support provided by 4.2BSD to provide a direct interface networks such as the ARPANET and/or Ethernet using SMTP (the Simple Mail Transfer Protocol) over a TCP/IP connection.

---

## REFERENCES

- [Crocker77] Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages*. RFC 733, NIC 41952. In [Feinler78]. November 1977.
- [Crocker82] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages*. RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.
- [Feinler78] Feinler, E., and Postel, J. (eds.), *ARPANET Protocol Handbook*. NIC 7104, Network Information Center, SRI International, Menlo Park, California. 1978.
- [Nowitz78] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems*. Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.
- [Schmidt79] Schmidt, E., *An Introduction to the Berkeley Network*. University of California, Berkeley California. 1979.
- [Shoens79] Shoens, K., *Mail Reference Manual*. University of California, Berkeley. In UNIX Programmer's Manual, Seventh Edition, Volume 2C. December 1979.
- [Solomon81] Solomon, M., Landweber, L., and Neuhengen, D., *The Design of the CSNET Name Server*. CS-DN-2. University of Wisconsin, Madison. October 1981.
- [Su82a] Su, Zaw-Sing, and Postel, Jon, *The Domain Naming Convention for Internet User Applications*. RFC819. Network Information Center, SRI International, Menlo Park, California. August 1982.
- [Su82b] Su, Zaw-Sing, *A Distributed System for Internet Name Service*. RFC830. Network Information Center, SRI International, Menlo Park, California. October 1982.

Chairperson: *Bill Tuthill*  
University of California, Berkeley

## **BIBFIND — A Bibliographic Retrieval System**

*James A. Moyer*

University of California, San Francisco  
San Francisco, CA 94143

A major problem for the research scientist is the storage indexing, and retrieval of bibliographic material. Members of our group, have used various mechanical methods of indexing reprints and articles. Such methods are time consuming, prone to error, require constant vigilance, and are difficult to delegate to someone else. The proliferation of small laboratory computers has made it possible to shift the burden of maintaining personal reference material to the computer. We have developed a system of programs to create, maintain, and search bibliographic reference files. This system, written in C, was developed on our V7-UNIX pdp11/34. The programs described below allow creation and access both to large common databases as well as smaller personal or project oriented reference lists. The system consists of five major modules that are used to create, edit, and search databases. The programs described are flexible enough to use either on a particular project (i. to collect and format references for a paper) or the more general tasks of indexing and searching a large collection of bibliographic material. The modules are described below.



---

## BIBFIND

### A Bibliographic Retrieval System

James Moyer  
Department of Pharmacology  
University of California San Francisco

A major problem for the research scientist is the storage, indexing, and retrieval of bibliographic material. Members of our group have used various mechanical methods of indexing reprints and articles. Such methods are time consuming, prone to error, require constant vigilance, and are difficult to delegate to someone else. The proliferation of small laboratory computers has made it possible to shift to the computer the burden of maintaining personal reference material. We have developed a system of programs to create, maintain, and search bibliographic reference files. This system, written in C, was developed on our V7-UNIX PDP 11/34. The programs described below allow creation and access to both large common databases as well as smaller personal or project-oriented reference lists. The system consists of five major modules that are used to create, edit, and search databases. The programs described are flexible enough to use on a particular project (eg. to collect and format references for a paper) or the more general tasks of indexing and searching a large collection of bibliographic material. The modules are described below.

Bibentry is the general input routine for the database. It is used either to add to or to create a new bibliographic file. Used

---

interactively, it prompts for input from the user and indicates the correct format for the entry. At the end of the input the user is given the choice of altering the completed entry or "submitting" the entry. Once an entry is submitted it is given a final format check and then placed either at the end of a private database or on a special output file for later updating of a public database. Each entry in a database is given a unique accession number that is used to unambiguously label a paper or book. On our system, entry is done chiefly by a secretary. The secretary is given a keywords underlined paper, who then enters the reference, and returns it marked with the accession number. A typical entry takes about a minute.

The bibkey module produces a series of indices into a database. These include an author index and a keyword index. Keywords are extracted from the title and keyword portions of the entry and are checked against a stopword file before inclusion in the index. An index entry consists of a coded word followed by offsets into the database. On our system the keys for a 500K character database require about 200K bytes of indexing.

Once an entry is in a database it can be changed or deleted with the bibedit module. Since the bibliographic files are in human readable form any editor can be used to edit entries. However this may become inconvenient with a file that contains many thousands of lines of text. Bibedit simplifies this task. It extracts an entry from the database and then forks any chosen editor (vi,ex,ed) to do the rest of the job. The edited entry is then

---

placed in a special location for later updating. Since creating a key can be time consuming, and the system and the database cannot be searched while a key is being produced, both `bibedit` and `bibentry` produce temporary files rather than touching the database itself. This scheme not only allows uninterrupted use of the system but provides an additional level of security and format checking.

Actual additions and changes to the database are done by the `bibupdate` module. This module checks the temporary files created by the edit and entry programs and makes a new database. Many checks are made to insure the integrity of the database. The old database is replaced only if all steps proceed without error. Any difficulties cause the update to be aborted and a diagnostic message to be mailed to the database administrator. On our system update is automatically invoked once a day at midnight to make any necessary changes to the common database. If changes are made `bibupdate` calls upon the `bibkey` module to generate the indices for the new database.

`Bibfind` is the general search and retrieval program for the system. It uses the files generated by the other modules to allow the user to selectively search for and format references in the database. Its simple syntax allows the casual user to perform author, date, and keyword searches. Several wildcard mechanisms are supported, as well as the relational AND, OR and NOT operators. Searches can be sequential; i.e., a command can operate on the output of a previous search. The use of index files

---

allow fast retrieval. A keyword search of our 500K byte database takes only two or three seconds when the system is lightly loaded. Output is either to the terminal or to a file. A number of available output formats may be selected and new output formats are easily defined. A "more" command is also available which allows output to be viewed a screenful at a time.

This system has become an indispensable tool for our group. By simplifying the more onerous tasks of indexing and filing, it has enabled us to cope with the information explosion in our field of medical research.

---

**This page intentionally left blank**

---

# COBOL Compiler Construction Experiences

## Using *lex* and *yacc*

*Robert E. Conant and Herbert G. Mayer*

Burroughs Corp.  
San Diego, CA 92127

COBOL? Why even mess with it in the first place? Like it or not, among programming languages, COBOL pays the bills for most computer vendors. The speakers described their experiences with using *lex* and *yacc* to build a COBOL 74 compiler. It was hoped that by using these tools they would be able to reduce development time, and produce a compiler that would be easier to maintain than compilers produced by traditional methods.

A number of problems were described, some each in the scanner and parser designs.

### The Scanner

COBOL permits any token to be split across lines! This, combined with COBOL's fixed-format source lines, is something that *lex* has trouble dealing with. The difficulty was resolved by providing a pre-pass which concatenates split tokens.

The number of keywords is enormous: 300 or so, compared with 28 or so with C. A first attempt might be to provide one *lex* rule for each keyword. Unfortunately, a scanner built this way tends to run for hours, only to terminate abnormally by running out of space. The solution was to put all 300 keywords into the symbol table, and to provide a single rule which is capable of recognizing all COBOL keywords as a general class.

Another problem is the PICTURE clause, which is too complex in general for the scanner to be able to recognize whether or not it's well-formed. The solution was to provide a series of rules which recognize a PICTURE string and leave the real work of verifying its correctness for the parser.

Significant blanks are another problem. For example, arithmetic infix operators are required to have significant blanks around them. To deal with this, each infix operator has two rules provided, one for the operator properly surrounded by blanks, and a second for the same operator without the blanks. The latter issues an error message, but returns the same token as the first, so that scanning can continue.

The COPY statement is COBOL's equivalent of *#include*. The problem is that it can appear ANYWHERE in the source text. This is dealt with by having the same pre-pass which concatenates split tokens also handle the COPY statement.

### The Parser

The first problem is that there is NO statement terminator. How do you recover gracefully after finding an error? How do you find the next statement? This problem was solved with a simple heuristic which involves using a global variable for communication between *yacc* and *lex*. A flag is set by *yacc* which tells *lex* to deviate from its normal mode of operation in which line terminators are ignored and tossed away, and instead watch for an end-of-line, and, when found, turn the error state off and continue scanning.

Other things that needed to be dealt with included 'dangling else' (and 'dangling other things') problems, and COBOL lists (some types of which can have zero elements, some require at least one element, and some require two or more!).

---

## COBOL COMPILER CONSTRUCTION EXPERIENCES USING LEX AND YACC

Robert E. Conant, Herbert G. Mayer  
Burroughs Corporation, Rancho Bernardo

### 1 Introduction

Although Cobol is an old and widely used programming language, it is relatively new to UNIX\*. Availability of good Cobol compilers is a prerequisite to increased commercial acceptance of UNIX.

This paper discusses the authors' recent experience at Burroughs in developing an ANSI 1974 Cobol compiler using Lex and Yacc under UNIX. The development systems were, initially, a PDP-11/70 and, later, a VAX-11/780. The discussion assumes that the reader is familiar with Cobol '74, Lex, Yacc, UNIX, the C programming language, and elementary compiler construction concepts. The following areas of experience are touched on:

- What are the compiler design goals relevant to the use of Lex and Yacc?
- How well did Lex and Yacc achieve the compiler design goals?
- What constraints of Lex and Yacc were encountered and how were they overcome?
- How well do Lex and Yacc fit the peculiarities of Cobol?
- What can be concluded about the suitability of Lex and Yacc for Cobol compiler construction?

The suitability of Lex and Yacc for Cobol compiler construction was not known at the start of compiler development. Cobol is an enormous language which originated before syntax-directed translation was widely used, and is often parsed in an ad hoc manner.

The incentive for attempting to use Lex and Yacc in constructing the Cobol compiler was:

- . Reduced development time, and
- . A more maintainable compiler.

---

\* UNIX is a Trademark of Bell Laboratories

---

## 2 Project Goals

Every compiler has many simultaneous, and sometimes conflicting, design goals. The following goals stand out as relevant to the use of Lex and Yacc:

1. Detect and report ALL lexical and syntactic errors. Recover gracefully.

2. Continue compilation, when possible, despite lexical and syntactic errors. Cobol has several rules which, when violated, do not usually affect the meaning of the program, such as those involving significant blanks, or the requirement that certain source constructs start in certain source line fields.

3. Minimize the amount of information the parser and scanner have to know about each other's state.

4. Allow the UNIX user to interact with the Cobol compiler in the way he is used to interacting with compilers of other languages on UNIX.

5. Use unaltered source versions of Lex and Yacc, if possible. Increasing Yacc table sizes is not considered an alteration in this sense. Modification of the output of Lex and Yacc is preferred over modifying Lex and Yacc themselves.



---

### 3 Lexical Analyzer

#### 3.1 Non-ANSII Accommodations to the UNIX Environment

The standard is vague on the issue of whether all lines of a Cobol source program must be of the same length. At any rate, it would be cumbersome to create fixed-length source lines using the UNIX editor programs, so our compiler accepts variable-length source lines.

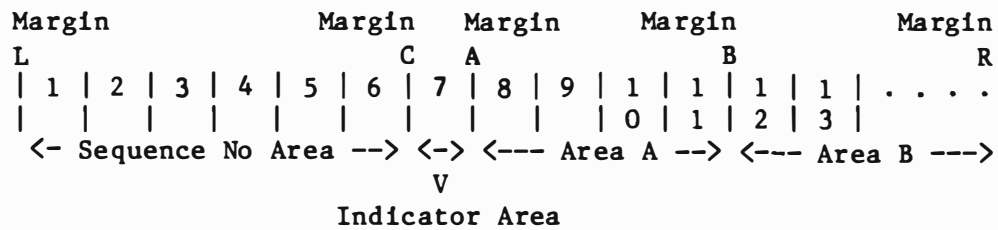
Although tab characters are disallowed by the standard, tabs are treated in a way similar to UNIX f77 compiler treatment. A tab character in any of the first six positions of a line signals the end of the Cobol sequence number and indicator areas of the line, and a blank indicator area is assumed. A tab in any other position is treated as a blank.

Consistent with the standard, our compiler expects only upper-case letters outside of non-numeric literals. Lower-case letters are converted to upper case when encountered, however, and an error message is displayed unless all such case error messages are silenced by a compilation option. Editing upper-case programs can be risky on UNIX. It is easy, when using the editor "ed", to issue a "W" command instead of the intended "w", thereby appending to, rather than rewriting, the source file.

#### 3.2 Recognizing Fixed-Format Source Lines

The first challenge in the lexical analysis of Cobol using Lex is discovering how to use a tool best suited to scanning variable-length source lines, possibly containing tab characters, terminated with the newline character, for scanning source lines of a language defined in terms of fixed-length lines comprising fixed-length fields.

Standard Cobol source programs consist of lines of implementation-defined length, each comprising fixed-length fields called "areas".



- . Sequence numbers must fill the Sequence Number Area.
- . Single-character indicators must occupy the Indicator Area.
- . Certain tokens must start in Area A.
- . Other tokens must start in Area B.

A solution is to view the Cobol source line as containing just three areas and to use unique Lex "start conditions" for each area:

- . Sequence Number Area (columns 1-6)
- . Indicator Area (column 7)
- . Areas A and B combined (columns 8-\n)

Selected Lex rules below illustrate the use of these three Lex areas:

```
%%
[\\n]                { BEGIN 0;          }
^[0-9]{6}            { BEGIN ind_area;    }
<ind_area>" "        { BEGIN area_A_or_B; }

.
.
.

<area_A_or_B>[0-9]+  { return(UNSIGNED_INTEGER); }
<area_A_or_B>[ \\t]+ { /* skip blanks and tabs */}
<area_A_or_B>[^ \\n] { printf("Illegal char\\n"); }
%%
```

Note the following about these rules:

- The Lex automaton is always able to recognize the newline character, and all other Lex rules are written so as not to consume the newline character.
- The 6-digit field after a newline character is the sequence field. Recognizing a sequence number activates a series of rules (with start condition <ind\_area>) which recognize the single character in the Indicator Area.
- Recognition of an Indicator Area character activates a series of rules which recognize tokens in Area A or B. These areas are combined here because tokens can cross the boundary between Areas A and B. Generally speaking, only

---

strings recognized in Areas A or B cause the scanner to return a token to the parser.

- Blanks and tabs up to, but not including, the newline character are skipped in Areas A and B. Then, recognition of the newline character starts the cycle again.

- Neither sequence numbers nor indicator area characters are returned by the scanner to be "seen" by the parser.

- The last rule is a "catch-all" rule, in that it will match all input, character by character, which was not matched by other rules above it. It must be the last rule in the Lex input. If no "catch-all" rule is present, unrecognized strings would be written to the user's standard output device, and would not be seen by the parser.

The task of determining whether a token recognized by the scanner appears in the correct area (A or B) is handled partly by the scanner and partly by the parser. The vast majority of tokens must start in Area B. Tokens which must start in Area A are:

1. Unsigned integers, but only when they are level-numbers.
2. Cobol words, but only when they are paragraph names.
3. The following reserved words:

- a. IDENTIFICATION
- b. PROGRAM-ID
- c. AUTHOR
- d. INSTALLATION
- e. DATE-WRITTEN
- f. DATE-COMPILED
- g. SECURITY
- h. ENVIRONMENT
- i. CONFIGURATION
- j. SOURCE-COMPUTER
- k. OBJECT-COMPUTER
- l. SPECIAL-NAMES
- m. INPUT-OUTPUT
- n. I-O-CONTROL
- o. DATA (but only before DIVISION)
- p. FILE (but only before SECTION)
- q. FD
- r. SD
- s. CD
- t. RD
- u. REPORT (but only before SECTION)
- v. PROCEDURE (but only before DIVISION)
- w. DECLARATIVES
- x. END (but only before DECLARATIVES)

The essential area checking scheme is this:

1. With every Cobol reserved word (they happen to be in the symbol table), store an indication of whether that keyword must appear in Area A or B, or whether it can legally appear in either area. The scanner checks those tokens which must appear in Area A or B.

2. Unsigned integers and Cobol-words recognized by the scanner can legally start in either Area A or B, depending on context, so the scanner can't check them for being in the proper area. However, every other token recognized by the scanner must start in Area B, and is checked for this.

3. The parser is responsible for checking the area of the following tokens, since only it has the required token context information:

- a. All unsigned integers
- b. All Cobol words
- c. Those few reserved words which can legally start in either area:
  - 1). DATA
  - 2). FILE
  - 3). END
  - 4). RECORD
  - 5). PROCEDURE

The parser has available two functions, "check\_a()" and "check\_b()" for this purpose. "check\_a()" issues an error message if the current token does not start in Area A, and similarly for "check\_b()".

```
%token DATA
int startcol = 1; /* current token's leftmost column */
%%
program: iddiv envdiv datdiv procddiv
;
datdiv : DATA { check_a(); } division period ...
        | error { printf("DATA DIVISION expected\n"); }
;
```

Function "check\_a" reports an error if the current token starts outside of Area A:

```
check_a()
{ if ( startcol < 8 || startcol > 11 )
  printf("Character-string must start in Area A\n");
}
```

Variable "startcol" is updated for each source string matching a Lex rule:

```
%{ int prevlen = 0; /* length of previous token */
```

```

%}
%%
[\\n] { prevlen = 0; BEGIN 0; }
<area_A_or_B>[+-]?[0-9]*"."[0-9]+ { col(yyleng);
                                     return(FIXED_POINT);
}

```

where function "col" is:

```

col(token_length) /* update curr. token start column */
int token_length; /* length (excluding terminating 0)
                   of current token */
{ startcol += prevlen; /* increment column by previous
                       token's length */
  prevlen = token_length; /* save curr. token's length */
}

```

### 3.3 Continuation of Source Lines

Because the scanner returns to the parser only those strings recognized in Areas A or B, line continuation is handled automatically except when a token spans multiple lines. In the extreme case, every token may be split into one character per source line, as follows:

```

123450      G
123451-     0
123452      T
123453-     0
123454      X

```

The standard says of line continuation: "A hyphen in the indicator area of a line indicates that the first nonblank character in Area B of the current line is the successor of the last nonblank character of the preceding line without any intervening space."

It is natural first to wonder if the scanner can perform this recognition and rejoining of multiple strings into a single token. Consider just the problem of recognizing a valid Cobol data-name split into pieces:

```

123451      0
123452-     CHANCE-
123453-     OF-SUCCESS
123454-     -1

```

The authors know of no way to write a reasonable number of Lex rules which will recognize that 0-CHANCE-OF-SUCCESS-1 is a Cobol word. In order to succeed, it would be necessary to be able to recognize any unsigned integer, signed integer, and any Cobol word from every leftmost substring of that integer or word. Worse yet, since the number of source lines on which a single token may be split is limited only by the number of

characters in the token, buffer overflow is a real likelihood. Lastly, any rules which could recognize a token split across line boundaries would either have to BEGIN other rules to recognize the embedded sequence and indicator fields, or would greatly complicate sequence checking of source lines.

An obvious solution is to have a pass over the source program which concatenates token parts before yylex sees the source. The user could have the option of omitting this extra source pass if he knows that the program contains no continuation lines, and yylex can easily check whether a program with continuation lines has slipped through. This is done by looking for '-' in the indicator area. The extra source pass would have replaced these '-' characters by blanks.

### 3.4 Recognizing Cobol Words and Reserved Words

Cobol was designed to read like English and, as a result, has many reserved words. The following list compares the number of reserved words in several popular algorithmic languages:

- 0 ANSI 1976 PL/I
- 28 1978 C
- 35 ISO Proposed PASCAL
- 47 ANSI 1978 FORTRAN
- 63 ANSI Proposed July 1982 ADA
- 300 ANSI 1974 Cobol

The first unsuccessful attempt at recognition of all 300 reserved words involved having one Lex rule for each. On the PDP-11/70, Lex had insufficient memory to execute to completion, so all reserved words were then placed into the symbol table at compiler initialization time. Since all Cobol reserved words obey the construction rules of Cobol-words, a single Lex rule recognizes both.

```
<area_A_or_B>[0-9]*[A-Z](?-[A-Z0-9])*  
{ ptr = lookup(yytext);  
  if ((found(ptr) && (ptr->keyword))  
      return(ptr->type);  
      /* return integer generated by Yacc %token */  
      else return(COBOL_WORD);  
      /* data-name, paragraph-name, etc. */  
}
```

After the project moved to a VAX-11/780, it was attempted again to place the reserved words in the Lex input, one rule per reserved word. Here are the results:

1. Lex function "acompute" exceeded some mysterious value 300, which was arbitrarily increased to 500. This allowed us to continue.

2. As the result of exceeding Lex array bounds on multiple

---

runs, the followed internal array size specifiers were placed in the Lex input:

```
%p 40000
%a 200000
%e 6000
%n 3000
```

Even with these internal array size specifiers, Lex still runs out of array space, and this writer ran out of patience.

3. Even if the internal array size specifiers are increased enough for Lex to complete execution, its execution time would probably be prohibitive. Lex took more than 2 hours clock time on the VAX to exceed array space with the internal array size specifiers above.

How large is the scanner produced by Lex? A scanner produced as a result of no input rules occupied just under 2K VAX bytes. A (useless) scanner which recognizes only the Cobol reserved words occupies 39.6K VAX bytes. With the reserved words in the symbol table, the scanner (plus actions) required to recognize the other Cobol constructs occupies 34K VAX bytes.

While all reserved words must be in the symbol table in order to be recognized, the presence of synonymous reserved words in Cobol means that the number of Yacc tokens required to represent all reserved words is fewer than the number of reserved words. For example, PIC and PICTURE are synonymous in all contexts, so a single token (take your pick) can be returned by yylex to yyparse when either reserved word is recognized. For the parser, having just one token, rather than two, to recognize means fewer parsing rules, and a smaller, faster parser.

### 3.5 Preventing Scanner Buffer Overflow

There is no way known to the authors to limit the length of tokens recognized by yylex, so it is possible to ruinously overflow the scanner's buffer, "yysbuf". On our version of Lex, the buffer size, YYLMAX is defined as 200, which is entirely adequate for even Cobol's longest lexical token, the non-numeric literal. Non-numeric literals require at most 122 bytes of the buffer.

The most straightforward method of insuring that yysbuf is not overflowed is to modify the "unput" macro of "lex.yy.c". This can be done by (automatically) editing "lex.yy.c" after each Lex execution, or by modifying Lex itself.

This is a serious limitation of Lex.

### 3.6 Recognizing Significant Blanks

Blanks are given significance in order to make the use of certain strings unambiguous:

- Around arithmetic infix operators.
- After period, comma, and semicolon separators.
- Before an opening pseudo-text delimiter.

Enforcing Cobol's rules concerning significant blanks is easy with Lex. Two methods are shown below, differing mainly in whether the scanner or the parser reports violations of the rules. Note that both methods are forgiving of violations of these "arbitrary" rules, and that significant blanks apply to the following tokens:

```
* " " (must be followed by a blank)
* " " ditto
* " ; " ditto
* " / " (must be surrounded by blank(s))
* " - " ditto
* " + " ditto
* " * " ditto
* " ** " ditto
```

Method 1 Lex rules: Report significant-blank errors from the scanner.

Note that rules such as " +" and "+ " are unnecessary since the white space rule reduces both cases to the "+" rule.

```
<area_A_or_B>"+" { printf("Blanks required around +0);
                  return('~+~');
                  }
<area_A_or_B>" + " { return('~+~'); /* correct form of + */ }
<area_A_or_B>[ \t]+ /* eat white space,
                  don't return to parser */
```

Method 2 Lex rules: Report significant-blank errors from the parser.

```
<area_A_or_B>"+" { return(BAD_PLUS); }
<area_A_or_B>" + " { return('~+~'); }
<area_A_or_B>[ \t]+ /* eat white space,
                  don't return to parser */
```

With Method 2, the parser would need the following rule:

```
plus: '~+'
      | BAD_PLUS { printf("Blanks required around +\n"); }
      ;
```



---

### 3.7 Recognizing Comments

There are two very dissimilar types of comments in Cobol: comment lines, which are trivially easy for Lex, and comment-entries (which appear only in the Identification Division) which are very difficult for Lex.

The Lex rules which recognize the two flavors of comment lines are:

```
<ind_area>"*"[^\\n]* { if (!comment_ok)
                        printf("Illegal placement\\n"); }
<ind_area>"/"[^\\n]* { if (!comment_ok)
                        printf("Illegal placement\\n");
                        /* emit form feed */
                      }
```

The variable "comment\_ok" is (unfortunately) a global variable which enables the parser to tell the scanner at what point in the source program comment lines may begin to appear. Its value, which is initially FALSE (0), is set as follows:

```
id_division: IDENTIFICATION division period
              { comment_ok = TRUE; }
              program_id period program_name period
              optional_author
              optional_installation
              optional_date_written
              optional_date_compiled
              optional_security
              ;
```

It is ironic that one of the scanner's most difficult problems is to ignore information, as in comment-entries. They are difficult because:

1. They have no starting delimiter.
2. They are terminated by a period separator, but that termination is ambiguous, since the comment-entry may contain non-terminating period-separators.
3. They may contain any characters.
4. They may be continued, but no indication of that continuation may be given in the Indicator Area.

The solution is too involved for discussion here but, just as with comment lines, comment-entries must be recognized up to the newline character by special Lex rules for this purpose.

### 3.8 Recognizing the Picture Clause

A record-description-entry in the Data Division may have the following optional clause:

$$\left[ ; \quad \left\{ \begin{array}{c} \text{PICTURE} \\ \text{PIC} \end{array} \right\} \quad \text{IS} \quad \text{character-string} \right]$$

Picture character-strings are formed according to a complex set of rules, and this set of rules can be viewed as a grammar of the language of all legal character-strings. Parsing picture character-strings in the scanner is simply too cumbersome to be practical, so the scanner should recognize picture character-strings without verifying that they are correctly formed. Then, a suitable character-string parser can be called, either from the Lex rule which recognized the character-string, or from the Yacc rule to which the character-string token was returned.

Since valid picture-strings can be confused with other lexical entities (9 is also a valid unsigned integer, A is also a valid data-name), it is necessary to have special Lex rules for the picture clause. Here are two different solutions:

Solution 1:

```
<area_A_or_B>[0-9]*[A-Z](-?[A-Z0-9])* { ptr=lookup(yytext);
    if ((f'      'ntr)&&(ptr->keyword))
    { if \_      type==PICTURE)
        BEGIN picture;
        .
        .
        .
        return(ptr->type);
    }
    .
    .
    .
}

<picture>"IS" { return(IS); }
<picture>[^ \t\n]+ { if (yytext[yy leng-1]=='.')
    /* if period ends string */
    yyless(yy leng-1);
    /* return period to input stream */
    BEGIN area_A_or_B;
    /* parse character-string
       here if desired */
    return(PIC_STRING);
}
```

The companion Yacc rules would be:

```

picture_clause: PICTURE optional_is picture_string
               |
               ;
optional_is    : IS
               |
               ;
picture_string: PIC STRING
               { /* parse character-string here */ }
               | error { printf("Picture-string expected\n"); }
               ;

```

#### Solution 2:

This solution differs from Solution 1 only in the way PICTURE or PIC is recognized. Note that the Lex rule which recognized PICTURE and PIC must precede the Lex rule which recognizes Cobol words, since if two or more rules match the same string, the earliest rule wins out. This solution has the disadvantage that processing common to all reserved words now excludes PICTURE and PIC. Note also that such legal data-names or paragraph names as PICTURE-PERFECT and PICKFORD will be matched only by the Cobol word rule. One token is returned for PICTURE or PIC, since they are synonymous in all contexts.

```

<area_A_or_B>("PICTURE"|"PIC") { return(PICTURE); }
<area_A_or_B>[0-9]*[A-Z](-?[A-Z0-9]) * { /* as above,
                                     excluding the test for PICTURE */
                                     }
<picture>[^ \t\n]+ { /* as above */ }

```

### 3.9 The COPY Statement

Cobol has a source text inclusion facility, the COPY statement, which presents implementation problems. A COPY statement can occur anywhere a character string or a separator may occur, but a COPY statement must not appear within another COPY statement. This last is not to be confused with a COPY statement occurring within an COPYed source text file, which is permitted. In addition, the COPY statement can cause all occurrences of designated character strings to be replaced before parsing by other character strings.

The main problem with implementing COPY is its ability to appear almost anywhere in the source program:

```
123456    ADD abc COPY file. TO def.
```

The potential ubiquity and complexity of the COPY statement are troublesome for the Lex-Yacc combination. Four methods of attacking COPY implementation are given below, in the order of discovery.

### Method 1.

The essence of this (hopeless) approach is to provide, for each separator token, parsing rules which allow a COPY statement to appear before or after the separator.

```
comma_separator : ',' copy_stmt
                | copy_stmt ','
                ;
copy_stmt       : COPY text-name
                | of_clause replacing_clause period
                ;
```

This approach would require separator non-terminals to appear in the parsing rules everywhere that a separator is permitted or required, for example:

```
envdiv : ENVIRONMENT blank_separator division
        blank_separator period blank_separator
        ;
```

Obviously, a large and slow parser would result from this approach, and readability of parser rules would be diminished.

### Method 2.

The scanner is adequate for handling the COPY statement's ubiquity problem, but not its complexity. The idea of this method is to recognize the COPY statement in the scanner. The following Lex rules (error checking omitted) show this approach:

```
COPY pathname [ { OF | IN } path_name ] ...

<area A or B>[ \t]+ "COPY"[ ]+ { BEGIN copy1; }
<copy1>[ ^ \t ]{1,14} { BEGIN copy2; /* UNIX pathname */ }
<copy2>[ \t]+ ( ("OF"[ \t]+) | ("IN"[ \t]+) | ("." " ") [ \t]+
               { /* OF|IN|"." */ BEGIN copy3; } )
<copy3>[ ^ \t ]{1,14} { BEGIN copy4; /* UNIX pathname */ }
.
.
.
```

This method would produce a very large scanner. Implementation of the full COPY statement would require a large number of Lex rules, especially once error detection and recovery rules are included.

### Method 3.

This method used Yacc rules to parse the COPY statement, but this time the COPY statement parser is separate from the Cobol program parser. This way, both the ability of the scanner always to recognize the beginning of a COPY statement, and the parsing power of Yacc are exploited. The Lex rule is simply:

```
<area_A_or_B>[ \t]+"COPY" { yyparse();
    /* call COPY parser */
}
```

Note that the COPY statement parser calls yylex recursively, and the input routine of yylex must be modified to handle a stack of input files. The Yacc rules for the COPY statement parser would parse only the COPY statement, and would begin as follows:

```
copy_stmt    : PATH_NAME of _clause replacing _clause
              period
              | error { printf("Invalid path-name\n"); }
              ;
.
.
.
```

Since the compiler now contains two functions named "yyparse", this name conflict must be circumvented.

#### Method 4.

The task of reading and concatenating the various COPY files can be performed by the same source pass that processes tokens split across source lines. This approach has the advantage of using the unmodified output of Lex.

#### 3.10 The "Decimal Point is Comma" Clause

The Environment Division contains an optional clause,

```
[ , DECIMAL-POINT IS COMMA ]
```

which, if present, reverses the roles of comma and period in fixed-point numeric literals. This is handled easily by Lex as follows:

```
<area_A_or_B>[+-]?[0-9]*"."[0-9]+ { if (dp_is_comma)
    printf("Period should be comma\n");
    return(FIXED_PT);
}
<area_A_or_B>[+-]?[0-9]*","[0-9]+ { if (!dp_is_comma)
    printf("Comma should be period\n");
    return(FIXED_PT);
}
```

---

Alternatively, both rules could be combined as follows:

`<area_A_or_B>[+-]?[0-9]*("."|",") [0-9]+`

It is slightly more difficult with the combined rule to check for conflicts between the decimal point found, and the state of "dp is comma", which is set by the parser upon recognizing the DECIMAL POINT IS COMMA clause.

### 3.11 Special Scanner Assists to the Parser

Because Cobol has no required statement terminators, the parser may choose to attempt resynchronization after an error in a statement by requesting a special token from the scanner the next time a newline is scanned. This is done on the premise that it is probable that statements are coded one per line. Normally, scanning a newline causes some bookkeeping code to be executed, but no token is returned. Here is how the special LINEND token is conditionally returned:

```
[\\n] { ...
    if (return_eol)
    { return_eol = FALSE;
      /* turn off the parser's request */
      return(LINEND);
      /* return line end token to parser */
    }
}
```

---

## 4 Syntax Analyzer

### 4.1 Implicit Assumptions

We assume the Cobol compiler will be hosted on a VAX or some UNIX system with comparable physical resources. One hundred kilobytes more or less in code or data space don't cause us any headache. Corollary: The same compiler will not run on a PDP-11/70.

We carry over a design decision from the developers of Lex and Yacc: Lex works as a subroutine under Yacc's control. Both share some common global variables. Corollary: We cannot break Lex and Yacc into two strictly sequential, independent passes.

### 4.2 Parsing Goals

The parser generated by Yacc has to recognize the full language Cobol ANSI'74. Consequently, the grammar input to Yacc must describe this language.

A software switch on the compiler invocation line directs the parser to recognize a Burroughs variant of Cobol, which is both a subset of and superset over ANSI Cobol. Except on few occasions we shall not mention this Burroughs variant of Cobol and instead shall only refer to the ANSI'74 Cobol language.

The second, equally important parsing goal is to reject every source string not in the Cobol language. Every syntax error should be properly flagged at the place of its occurrence. So both goals together mean: "Compile the whole language and catch all errors."

The latter alone is known to be a big task. It came therefore as a big surprise to us that Yacc would detect all static syntax errors without any extra provisions for it. All that is needed is to provide Yacc with a grammar which accepts just the language defined and nothing else. As soon as an illegal source string is read, the parser finds no legal rule to reduce and panics. It expresses the panic by emitting "Syntax error".

But as always in life you only get what you pay for (or less). The error detection you get for free in Yacc is not very useful. No mention is made about the exact nature of the error. We needed a more rigorous scheme of error detection and correction that would minimize the emission of these Yacc default error messages. When "Syntax error" appears on the screen, the parser should be pretty much lost.

The third main goal is to minimize the number of non-terminal

---

symbols and the number of grammar rules. As we shall see, this modest set of goals causes quite a conflict of interests.

#### 4.3 What is particularly easy with Cobol and Yacc

Statement identification is trivial in Cobol, since all statements start with a different reserved keyword. Even though a few of the arithmetic verbs such as ADD and DIVIDE come in different variations, simple factorization of rules yields a LALR(1) grammar. The large number of keywords may be perceived as a curse somewhere else, but as far as parsing is concerned it is a blessing.

Cobol often specifies upper bounds on syntax constructs. The dimension of tables, for example, may not exceed three; the maximum nesting of Cobol structures is 49. Consequently it is only possible to specify at most three indices and at most 49 levels of qualification for a data name. If a data structure is needed to temporarily hold source tokens such as the individual identifiers of a qualified name, then a reasonable upper bound for the data structure is implied by the nature of the language definition. This is explained in more detail elsewhere in the document with the SAVEID[49] mechanism.

Left recursive rules are allowed in the grammar. If one would dare to use the same construct in a recursive descent parser, the result would be catastrophic. Yacc knows no such limit. Also Yacc allows the forward referencing of names, something quite comfortable when contrasted with typical one-pass compilers.

Explicit prioritization of operators or rules in Yacc grammars is an equally powerful scheme. It allows the writing of a quite minimal grammar for expressions, which otherwise would require a complex set of rules. Note that this method of making rules unambiguous also is consistent with the design goal of keeping the number of rules small.

Depending on the chosen parsing strategy, one particular Cobol construct could be hard to analyze. We refer to abbreviated combined relation conditions, short ACRCs. These turn out to be no parsing problem when using Yacc. Assuming that rules exist for conditions, and rules are provided for combining these by AND and OR operators, then ACRCs only require the addition of one single production for optional relational operators.

The finding and processing of the corresponding expression subtree is not different from that found in other parsers.

Cobol source examples:



---

```
IF A > B - 5 OR C + 3           same as:
IF A > B - 5 OR A > C + 3
```

or:

```
IF A + 1 > B + 2 OR C + 3 < D + 4 AND = E   same as:
IF A + 1 > B + 2 OR C + 3 < D + 4 AND C + 3 = E
```

#### 4.4 Hard Problems and Our Solutions

We should not be surprised that this section is noticeably larger than the previous one, where we list the easy things. Both Cobol and Yacc pose some "hard nuts to crack" for the compiler writer.

As it turns out all three parser design goals are nontrivial: recognizing full Cobol; creating reasonable error messages and graceful error recovery; and minimization of grammar rules. Graceful error recovery is quite a challenge in itself, since Cobol lacks required statement terminators.

##### Error Recovery

We have developed some heuristics, based on the following assumption: source statements often terminate at the end of a line. In other words, it is rare that several statements are written on the same line. This does not mean all linends are statement ends, since statements may span multiple lines. But the above assumption represents a reasonable approximation.

Therefore, as soon as a syntax error is detected, the parser informs Lex about this via an error switch. Lex in turn no longer silently swallows linends but returns a special LINEND token to the parser, eagerly looking for just that. After having scanned one linend, Lex flips the error switch back to normal.

The following kind of rule is used for error recovery:

```
rule      : required_token rest
          | { return_eol=TRUE; } error LINEND
          ;
```

##### Optional Postfix

Two "Cobol specials" don't directly cause a problem but a nuisance in the form of Yacc-time error messages. Both caused us some worries initially, until we learned to just "live with it."

These "specials" are "lists of elements" at the end of a rule or "optional rules" also at the end of more encompassing rules. Yacc immediately flags these with shift/reduce errors. A problem arises, however, if one tries to eliminate the error messages by transforming the grammar into an equivalent, unambiguous LALR(1) grammar.

This type of complication does not show up in languages requiring statement terminators.

Two examples are given from our listing:

```
addstart      : ADD source_list
               ;
add_stmt      : addstart TO sink_list
               | addstart GIVING sink_list
               ;
sink_list     : sink_list sink
               | sink
               ;
comp_stmt     : id_ref opt_round assign_op exp opt_on_size
               ;
opt_on_size   : on_size_error
               ;
```

There is no real solution to the shift/reduce conflict for lists at the end of a rule. We just assert that Yacc prefers the shift over the reduce action and then ignore the conflict message.

#### Floating List End

A special case of the list problem above shows up in the following context: Most arithmetic verbs like ADD and MULTIPLY may end with lists of datanames. The list elements may, but need not, be separated from one another by comma or semicolon. Two samples are given below:

```
ADD X, 7 GIVING A OF B, C OF D OF E( I ), F OF G      same as:
ADD X 7 GIVING A OF B C OF D OF E( I ) F OF G
```

The end of such a list is marked implicitly by the start of the next statement or the beginning of the next paragraph. Paragraphs, however, also start with an identifier, which is the paragraph name, which is comparable to a label in other languages. Once the parser sees this identifier, it misinterprets it for yet another member of the current list of datanames, and it is too late. One symbol too much has been

---

swallowed.

Luckily the rules of the Cobol language cooperate here to solve the conflict. A paragraph name must start in a certain range of source columns, called AREA A, which is mutually exclusive from legal source columns for identifiers, named AREA B.

Lex can use this restriction by returning a PARAGRAPH\_NAME token in the one case and an IDENTIFIER token in the other situation. Thus the parser knows where to stop. The solution lies in the wise use of language restrictions more than in the interaction between Yacc and Lex.

### Optional Semicolon

Just because the semicolon is optional, it must not be allowed to fall through the cracks. Instead Lex must return a token, such that the parser can check whether it occurs in legal contexts before it throws the token away. A minor nuisance is created by various shift/reduce error messages which are generously ignored by us. This is nothing but a special case of a previously mentioned trouble spot: Optional constructs at the end of rules.

### Minimal Lists

Some lists in Cobol may have zero or more elements while others must have at least one element. There are two standard parser solutions to guarantee at least one element. One is to count the list elements and return the number in a \$ parameter automatically provided by Yacc. The user of the list rule then has the responsibility of checking the number in a semantic action. Counting is straightforward but uses a valuable resource, the one and only \$ parameter per rule. Counting is demonstrated below:

```
list      : list element { $$ = $1 + 1; }
          | { $$ = 0; }
          ;
```

The second approach for verifying the minimal number of elements in a list is to write a more complex grammar, also shown below:

```
list1     : req_element list
          ;
req_element : element
```

```

| error { need_one(); }
;
list      : list element
|
;

```

#### Long Common Prefix

When two distinct syntax constructs have a long common prefix, (long for a LALR(1) grammar means more than one lookahead token,) there may still be an equivalent LALR(1) grammar for this language. Factorization is needed here, as demonstrated with an example below:

```

A IS NOT GREATER THAN 5      /* a relation condition */
A IS NOT ALPHABETIC          /* a class    condition */

```

The difficulty here lies in the fact that the first kind of condition requires a second operand after the operator, to be tested with the first one for a relation to hold, while the class condition has only one operand. The keyword IS is optional, also called a noise word, while the NOT operator is optional but with a definite meaning.

The solution comes by factorizing, but a cascade of additional rules is being generated. The real issue here lies a bit deeper still. If Cobol would not permit certain abbreviations, referred to above as ACRCs, then the expression parser could at least EXPECT a relational or class condition operator and emit errors if not found. Unfortunately this would be incorrect. In combined relation the lack of a relation operator is perfectly legal after any of the boolean operators AND or Or. An appropriate implicit first operand and An implicit relational operator, however, must be available.

We demonstrate a solution by factorization here in two stages, first for a subset of Cobol, in which class conditions may not have either the IS noise word or the NOT operator.

```

basic_class : ARITHMETIC
            | NUMERIC
            ;
basic_relop : GREATER
            | LESS
            | EQUAL
            ;
req_relop  : basic_relop
            | error { give_me_relops(); }
            ;
relop      : IS not_req_relop
            | NOT req_relop
            | basic_relop
            ;
not_req_relop : NOT req_relop
              | req_relop
              ;
expr         : expr relop expr
              | expr basic_class
              *
              *
              ;

```

The above grammar is sufficient for the mentioned Cobol subset, since the class condition rules do not interfere with relations. Below is a grammar for Cobol's conditions without the simplifying restriction for class conditions; i.e. ALPHABETIC may also be preceded by the noiseword IS and may be negated by NOT.

```

basic_class : ARITHMETIC
            | NUMERIC
            ;
basic_relop : GREATER
            | LESS
            | EQUAL
            ;
expr        : expr is_clauses
            | expr not_clauses
            | expr basic_relop expr /* trouble spot */
            | expr basic_class
            *
            *
            ;
is_clauses  : IS pos_class_rel
            | IS NOT pos_class_rel

```

```

not_clauses : NOT pos_class_rel
;
pos_class_rel : basic_relop expr %prec RELOP
| basic_class
| error { relation_after_is_not(); }
;

```

## Cobol Structures

A delicate parsing problem comes with Cobol structures, also named records. Data declarations including structures must start with an unsigned number. These may range from 01 to 49 and include three more values 66, 77, 88 for special purposes. The value of the unsigned number, however, does not give a clue to the parser, even though it is the only distinguishing information for several symbols yet to come.

For a while we tried to have Lex return special source tokens for 01, 66, 77, and 88. Instead of providing the parser with UNSIGNED\_INT, it produced tokens like O ONE, SIX SIX. This turned out to be a dead-end approach. Since, on the other side, Lex should have no parser context information, it cannot turn off this state in the procedure division where a literal 01 is no different from a literal 10000, except in its numerical value.

This approach was quickly abandoned and replaced by the grammar given below.

```

dcl_start : UNSIGNED_INT idf_filler
;
idf_filler : FILLER
| IDENTIFIER { enter_symtab( IDENTIFIER ); }
| error { expected( "IDENTIFIER or FILLER" ); }
;
dcl : dcl_start data_type req_ender
| dcl_start ENDER
| dcl_start renames req_ender
| dcl_start val_through through_list req_ender
;
req_ender : ENDER
| error { expected( PERIOD ); }
;
renames : RENAMES req_idf throughpart
;
val_through : VALUE opt_is ...
;
.

```

---

## Greibach Normal Form

Every Context Free (CF) grammar can be transformed into an equivalent CF with certain nice properties. One such property is to have only rules of the form:

```
nt      : TOKEN ntlist
;
```

where TOKEN is a terminal symbol in the grammar's vocabulary, and ntlist is any number of non-terminals, including zero. This form is called Greibach Normal Form (GNF).

Yacc's pseudo rule "error" makes sense in only one place, and that is at the start of a rule. Whenever the grammar under consideration has productions of the form

```
ntl     : T1 nt2 T2 nt3 T3
;
```

where T1, T2, and T3 are required terminals, the design goal for precise error diagnostics cannot be met. A missing T2 or T3 in the source stream would only cause Yacc's unpleasant "Syntax error" to appear. Therefore rules of the above form must be rewritten into GNF, as shown by example below.

```
ntl     : T1 nt2 req_t2 nt3 req_t3
        | error { give_me( T1 ); }
;
req_t2   : T2
        | error { give_me( T2 ); }
;
req_t3   : T3
        | error { give_me( T3 ); }
;
```

The above is a perfect paradigm for situations where striving for precise error diagnostics collides with the principle of keeping the number of rules small.

Our overall Cobol grammar is not strictly in GNF, but we use GNF whenever the language requires a terminal symbol.

## No Backtrack with Yacc

Yacc allows the embedding of semantic actions in the grammar. These actions are treated as pseudo rules and are not different from other rules. There exists, however, one subtle and important deviation from this. If an action precedes the first terminal or non-terminal of a rule, then, even if the rule is

---

reduced during the parsing process, the semantic action's pseudo rule is effective AFTER shifting the first symbol.

```
nt          : { switch_on(); } TERMINAL { switch_off(); } rest
             | some_other
             ;
```

The action `switch_on` is executed after `TERMINAL` has been shifted. In such a case it is not possible to communicate with Lex for the purpose of doing something special with the `TERMINAL`, since it would be too late. The grammar rule above is indeed equivalent to the next rule below.

```
nt          : TERMINAL { switch_on(); switch_off(); } rest
             | some_other
             ;
```

The two desired semantic actions may turn out to be unusable.

#### Global Variables Considered Harmful

One of the weakest spots in the interaction between Lex and Yacc is the mechanism of communicating. The main channel for passing and returning information is via the use of global variables.

Some language requirements can be checked either by setting and testing global variables, or by writing more rules, thus increasing the physical grammar size. As an example we list the `IF` and `NEXT SENTENCE` statements of Cobol, the latter of which may only appear when embedded in an `IF` statement. One way to assure this is to keep track of the static nesting of `IF`s via a global variable. Another way is to write a more complex grammar, disallowing `NEXT SENTENCE` in one context and allowing it in the other.

In an effort to reduce the number of global variables, we have introduced a catch all global array named `SAVEID`, which can hold up to 49 remembered values. In addition there are quite a number of elementary globals such as the `if_nest` counter.

Typically `SAVEID` is used for storing lists of identifiers when operations on these identifiers, like table lookups, are done long after the lexical token has disappeared. Again, "long" means one or more source tokens later.

The information stored in `SAVEID` has a lifetime much shorter than that of the global variable itself, being a source of confusion sometimes. It is easy to forget that all `SAVEID` data are lost as soon as the next list in the source program is



---

parsed, thus obsoleting the previous list.

#### 4.5 Wish List for Yacc Improvements

##### Conditional Parsing

As was mentioned in the section about Cobol structure-parsing, the actual value of a token has no impact on the parse process. The value is, however, available in a global variable which is shared between Lex and Yacc. "It would be nice" to extend Yacc to allow rules like:

```
dcl      : UNSIGNED_INT == 01 tail1
         | UNSIGNED_INT == 66 tail66
         | UNSIGNED_INT == 77 tail77
         | UNSIGNED_INT == 88 tail88
         | UNSIGNED_INT all_other_declarations
         ;
```

##### Disallowing Tokens

Another useful operation would be to "Shift On Any Token Except", expressed by the ^ operator. This would allow a shift on all terminal symbols except one and would be expressed as in the sample grammar below:

```
skip_dot : PERIOD recovered
         | ^ PERIOD skip_dot
         ;
```

##### Miscellaneous Extensions

A cross reference phase would be quite helpful, together with a listing and linenumbers.

Also the passing of input parameters to non-terminals in addition to the output \$ parameters was dearly needed on several occasions. The input parameter could then be interrogated in semantic actions, just as the output parameters \$ can.

##### Table Size Specifications as Yacc command Line Options

This would make it unnecessary to have a private version of Yacc. Another approach to table size specification could be the use of Yacc directives in Yacc's input file, in the style of Lex.

#### 4.6 Yacc Statistics

---

The last few lines of the verbose output of Yacc are included here. They give some static information about the analyzed grammar.

368/450 terminals, 588/650 nonterminals  
1281/1500 grammar rules, 1941/2000 states  
35 shift/reduce, 9 reduce/reduce conflicts reported  
588/650 working sets used  
memory: states,etc. 9088/12000, parser 2914/12000  
1338/1600 distinct lookahead sets  
3177 extra closures  
3031 shift entries, 173 exceptions  
1362 goto entries  
1403 entries saved by goto default  
Optimizer space used: input 8200/12000, output 3628/12000  
3628 table entries, 1201 zero  
maximum spread: 603, maximum offset: 1936

#### 4.7 Remaining S/R R/R Conflicts

The majority of shift reduce errors is caused and cannot be eliminated by lists at the end of other rules. These and optional rules at the end account for a total of 29 shift/reduce errors, where the optional semicolon is represented 13 times.

Also the optional ELSE clause in IF statements causes a shift reduce error. This is not counted as one of the 29 cases above, since it is possible to write an unambiguous grammar for the IF statement. We chose not to do so to keep the grammar small. Yacc's default action is just what is needed anyway, so the generated parser is correct in this regard.

The remaining conflicts are caused by "temporary" solutions which are known by us to be imprecise, and for which we shall supply a more complete and correct grammar in the future.

#### 4.8 Yaw, Yet Another Wish

Many of the experiences we have gained during our growing friendship with Lex and Yacc are valuable but have been costly in terms of time. We wish that at the outset of our Cobol compiler project we had had a paper like the present one at our disposal. We feel this would have saved us time and frustration.

One of our motives in writing this paper was to make our experiences available to other compiler writers who were taking a similar approach. Our true audience is therefore necessarily small; but should there be any member in this set, we believe this implementor will benefit from our small contribution.

---

## 5 Conclusions

An ANSI '74 Cobol compiler can be constructed using Lex and Yacc. A PDP-11/70 has insufficient memory for Yacc to process the large Cobol grammar rules, but a VAX-11/780 has sufficient (virtual) memory. It is necessary to increase Yacc table sizes by modifying "#define" statements in Yacc. Yacc's execution time is acceptable.

Cobol has too many reserved words to be recognized directly by Lex. A solution is to store all reserved words in the symbol table. There are few instances in which the scanner produced by Lex must know the state of the parser. Lex's execution time is acceptable. A serious defect of the scanner produced by Lex is that overflow of the scanner's input buffer cannot be predicted or prevented, given pathological input.

The Lex/Yacc combination accomplishes all our design goals, with the least success in the areas of parser error recovery, and minimization of knowledge of each other's state.

The performance of the compiler produced by Lex and Yacc remains unevaluated. Even if the scanner is later handcoded for increased performance, it is believed that using Lex initially is valuable for specification of a hand-coded scanner.

---

## Development of *refer*: Bug Fixes and Enhancements (or (unofficially) “Refer Madness”)

*Bill Tuthill*

University of California, Berkeley  
Berkeley, CA 94720

*Refer* is a bibliographic tool which, by itself, doesn't really fit the needs of a university community. Accordingly, a number of programs were created to make a bibliographic system easier to use. These include tools for creating and extending bibliographies, sorting them by arbitrary keys, formatting them (using *nrdf*), creating indices, and so forth.

The `—ms` macros were revised somewhat, with a resulting increase in speed.

*Refer* itself was enhanced. Changes were made and options added so that it would provide the following new features [I couldn't write fast enough to get them all down; sorry.]:

1. Insert square brackets around references.
2. Permit setting of footnote numbers.
3. Sort arbitrarily large numbers of references without dumping core.
4. Turn off the automatic searching of `/usr/dict/papers`.
5. Produce endnotes instead of footnotes.
6. No longer behave anomalously in the presence of trailing white space.
7. Permit 'corporate authors' (i.e. names are sorted by first word of author's name).
8. Collapse ranges of consecutive numbers (i.e. M-N instead of M,M+1,...,N-1,N).
9. Support pre-1900 and post-1999 dates.

---

## Development of Refer : Bug Fixes and Enhancements

*Bill Tuthill*

Computing Services  
University of California  
Berkeley, CA 94720

ucbvax!g:tut  
g.tut@berkeley

### Introduction

The **refer** program was deleted from (or not included on) System 3, presumably because of bugs or portability problems. It should be saved because it is a convenient tool for composing papers and compiling bibliographies. Furthermore, it comes with software for making inverted indexes, which allow retrieval of a reference from a 32 megabyte database in only 1.7 seconds of cpu time (on a PDP 11/70). The indexes generally use between 10-20% as much disk space as the database.<sup>1</sup>

The **Refer** system uses variable-length data, with percent/key-letter field labels, and a blank line to separate records. Here are two sample bibliography entries:

```
%A Bill Tuthill
%T Refer — A Bibliography System
%I Computing Services
%C Berkeley
%D December 1982
%O UNIX 4.3.5

%A M. E. Lesk
%T Some Applications of Inverted Indexes
  on the Unix System
%B Unix Programmer's Manual
%V 2A
%I Bell Laboratories
%C Murray Hill, NJ
%D January 1979
```

Fields may be continued on as many lines as necessary, as in the title of the second document. This method of storing bibliographic information is efficient in terms of space, and permits fields to be entered in any order. However, the number of unique keys is limited by the number of printable ASCII characters.

---

<sup>1</sup> Mike E. Lesk, "Some Applications of Inverted Indexes on the Unix System," in the *Unix Programmer's Manual*, Bell Laboratories, Murray Hill, NJ, January 1979.

---

The **refer** program is a preprocessor for **nroff/troff**, like **eqn** and **tbl**, except that it is used for literature citations, rather than for equations and tables. Given incomplete but sufficiently precise citations containing the author's name, some keywords, or the date, **refer** finds and delivers references contained in a bibliographic database. For example, the above footnote was specified with:

```
.[
Lesk Inverted Indexes
.]
```

The complete references are formatted as footnotes, numbered, and placed either at the bottom of the page, or at the end of a chapter. Thus, writers need not worry about specific publication information, or about numbering or labeling.

### Component Programs

Here are the programs that make up this bibliography system. The **refer** program is from Bell Laboratories; the others were pieced together at UC Berkeley. Even though many of these programs are trivial, they make **Refer** easier to use than it would be by itself.

- addbib** - create or extend bibliography
- sortbib** - sort bibliographic database
- roffbib** - format bibliographic database
- indxbib** - do inverted index to bibliography
- lookbib** - find references in a bibliography
- refer** - find and insert references in papers

The **addbib** program is for creating and extending the bibliographic database; **sortbib** sorts the bibliography by author and date, or other selected criteria; and **roffbib** runs off the entire database, formatting it not as footnotes, but as a bibliography or annotated bibliography. Once a full bibliography has been created, access time can be improved by making an index to the references with **indxbib**. Then, the **lookbib** program can be used to quickly retrieve individual citations or groups of citations. Creating this inverted index will speed up **refer**, and **lookbib** will allow you to verify that a citation is sufficiently precise to deliver just one reference.

Only **addbib** and **sortbib** have to be C programs (**lookbib** was just recently rewritten in C). **Addbib** is a simple data entry program with an editor escape. **Sortbib** reads a bibliography file for sort keys, then uses disk seeks and reads to retrieve records in the proper order. The remaining programs are simple shell scripts. Here is **indxbib** (which used to be called **pubindex**):

```
:    indxbib sh script
if test $1
then
    /usr/lib/refer/mkey $* | /usr/lib/refer/inv /tmp/$1
    mv /tmp/$1.ia $1.ia
    mv /tmp/$1.ib $1.ib
    mv /tmp/$1.ic $1.ic
else
    echo 'Usage: indxbib database [ ... ]
    first argument is the basename for indexes
    indexes will be called database.{ia,ib,ic}'
fi
```

Both **mkey** and **inv** should reside in **/usr/lib/refer**. They are from Bell Laboratories, and are available on the V7 distribution tape. The first pulls out selected indexing terms, while the second constructs an inverted index.

---

Here is **lookbib** :

```
: lookbib sh script
if test $1
then
    /usr/lib/refer/hunt $1
else
    echo 'Usage: lookbib database
    database must have indexes built by indxbib
    finds citations specified on standard input'
fi
```

Again, **hunt** should reside in `/usr/lib/refer`, and is part of the V7 distribution. It uses the inverted index generated by **inv** to deliver references corresponding to queries. The **roffbib** shell script is somewhat more complicated, mostly because it requires argument parsing:

```
: roffbib sh script

flags=
abstr=
macro=-mbib

for i
do case $1 in
    -[onsrT]*-[qeh])
        flags="$flags $1"
        shift ;;
    -x)
        abstr=.X
        shift ;;
    -m)
        shift
        macro="-i $1"
        shift ;;
    -*)
        echo unknown flag: $1
        shift
    esac
done
if test $1
then
    refer -al -B$abstr $* | nroff $flags $macro
else
    refer -al -B$abstr | nroff $flags $macro
fi
```

The `-m` flag of **roffbib** allows the user to specify a custom set of macros. The `-B` flag of **refer** indicates that bibliography records are separated with a blank line rather than with `.[. ]` pairs.

### Refer Bug Fixes

About 20 bugs in **refer** itself (mostly affecting output) were fixed; some of these were misfeatures rather than true bugs. Additionally, **refer** was revised to reduce **lint** errors, which should make it easier to port than before. The V7 **refer** will not compile on VAX without modification, for instance, and the

---

VAX version does not work properly on a PDP. One option (the -n flag) is not upwardly compatible; all others are, even though -k and -l should have been reversed. A quick summary follows.

- There was no easy way to run off a stand-alone bibliography. Users had to hand-edit their bibliography, placing .[.] pairs between references. Now **refer -B** inserts .[.] pairs automatically.
- Footnotes could not start with any number except 1, and could not go higher than 200. Now the -f option sets the footnote number, and footnote numbers can go up to 500.
- The -s (sort) flag caused a core dump when too many references were sorted at once. However, **sortbib** can alphabetize an arbitrary number of references, using disk seeks and reads.
- The default system bibliography, /usr/dict/papers, was useless for most people, so -n should have been automatic. Now, if there is a REFER environment variable, the default bibliography is not searched.
- With the -e and -s options, no more than 2000 characters of reference pointers were allowed. This limit has been raised to 10,000 characters, which seems large enough.
- Trailing blanks on the %A line messed up reversing and sorting, and trailing white space caused other anomalous behavior. Now trailing white space is ignored.
- There was no way to deal with corporate authors, or with authors having Japanese or Arabic names (where the surname comes first). Now there's a %Q field for quorporate authors, which is treated just like the %A field, except that it is not reversed for sorting.
- There was no way to deal with books in a publication series. Now there's an %S field for Series title, which is printed after the book title.
- The Natural and Social Science format was hard to produce. The -k flag produced [Kernighan1978a] rather than (Kernighan and Ritchie, 1978). There's now a -S option for producing this style of references.
- Large ranges of footnote numbers had a comma between every number. Now ranges of consecutive numbers are collapsed with a dash.
- Dates earlier than 1900 should be recognized as valid search strings; perhaps page numbers should be recognized as well. Currently numbers containing three or more digits are indexed; this includes dates before 1900, but doesn't include page numbers below 100.
- Periods and commas can go before or after a reference signal (macro-controlled), but ; : ? and ! always came *before* the reference signal. A new -P option will place punctuation marks after the signal; by default all punctuation comes before.
- The macros in /usr/lib/tmac/tmac.srefs were inconsistent between reference types and contained several errors. These macros were extensively revised; this alone eliminated many complaints. The new macro package, available as -mx, is upwardly compatible with -ms, and about twice as fast starting up. It also contains features for doing numbered footnotes and endnotes, tables of contents, even and odd headers and footers, and extended accent marks.



---

**This page intentionally left blank**

---

# **RAPID: A Tool for Building Interactive Information Systems**

*Anthony I. Wasserman and David T. Shewmake*

University of California, San Francisco  
San Francisco, CA 94143

How does one go about designing interactive information systems? It's important, of course, that the user-program dialogue be friendly and usable, and that the software be reliable and reasonably efficient. The end-user, however, isn't really concerned about the details of the underlying software. [Good quote: "But, let's face it — Most of the users don't care what programming language you use. They don't CARE whether or not you've used 'GOTO' statements. They simply want the program to work for them when they want it to work. And they want it to be easy to use, and use terminology that's comprehensible to them, and not muck up their screens with a lot of low-level error diagnostics."]

In general, an interactive information system can be characterized by: (1) a user-program dialogue, (2) an underlying database, and (3) a set of operations on the database (e.g. queries, modifications) which are effected by the user-program dialogue. User Software Engineering (USE) is a methodology developed for the purpose of building systems of this sort in an orderly and rational fashion. The emphasis is on understanding the methodology involved in designing interactive information systems, and designing tools to fit that methodology, rather than the other way around.

One important goal is to ensure that the people who will ultimately be using the software be involved in its design from the very start. After some initial analysis and modeling, and identification of relevant user characteristics and skills, a decision is made — very early on — about what the user-program dialogue is going to look like.

RAPID is a tool which permits early prototyping by supporting a language which can be used to define nodes and transitions between them. A transition is triggered by a user input, and transitions have particular actions associated with them. RAPID's 'transition diagram interpreter' can thus be used to create a prototype system which links a description of the dialogue with the database and with a set of actions on that database. The ability to do fast prototyping permits effective user involvement in the design of the overall system and allows designers and users to experiment with different user interfaces.

Moreover, the ability to experiment with alternative user interfaces to the same set of operations provides an opportunity to evaluate (both subjectively and objectively) users' performance with different types of user-program dialogue, and thereby make some judgments about what sort of user interface is most appropriate.

Rapid prototyping also gets users actively involved in thinking about what the system will ultimately look like, with the result that they're apt to start thinking about features or functions that might not otherwise have initially occurred to the developers, or, for that matter, to the users themselves.

---

## **RAPID: a Tool for Building Interactive Information Systems**

**Anthony I. Wasserman  
David T. Shewmake**

**Medical Information Science  
University of California, San Francisco  
San Francisco, CA 94143**

User Software Engineering (USE) is a methodology and development environment to support the specification, design, and implementation of interactive information systems. A key aspect of the USE methodology is the rapid construction and modification of prototype versions of the system.

There are several reasons for building such a prototype:

- (1) it enables the user to evaluate the interface in practice and to suggest changes to the interface;
- (2) it enables the developer to evaluate user performance with the interface and to modify it so as to minimize user errors and improve user satisfaction;
- (3) it facilitates experimentation with a number of alternative interfaces and modification of interfaces.
- (4) it gives the user a more immediate sense of the proposed system and thereby encourages users to think more carefully about the needed and desirable characteristics of the system.
- (5) it reduces the likelihood of project failure.

Rapid Prototyping of Interactive Dialogues (RAPID) is a tool that has been built for this purpose. The underlying concept is that the user/program dialogue and the associated actions can be specified using augmented state transition diagrams.

A transition diagram is a series of nodes and directed arcs. An arc is selected and traversed based upon user input. An action, such as updating a data base, may be associated with traversal of the arc. Upon traversing the arc, the terminal is then in the state represented by the node at the end of the arc.

RAPID automates the definition of nodes and arcs. Output messages may be associated with nodes, and actions may be associated with arcs. The message facility is screen-oriented, so that full cursor control is available along with output. (It uses the BSD curses package and termcap database to provide specific terminal information.)

Varying levels of functionality may be used in RAPID, from a pure dialogue to a fully functional system. Functionality is achieved through action calls to Unix functions. When the action calls include database management, the resulting system provides conversational access to a database. (The Troll/USE relational database management system is another tool in the User Software Engineering environment.)

Input to RAPID consists of one or more conversations, each representing a transition diagram, along with optional actions. Each conversation may have four types of statements:

- (1) Diagram name statement -- identifies the diagram, its entry node and exit node.
- (2) Variable definition statements -- permits the use of names to describe strings of alphanumeric or numeric characters.
- (3) Node definition statements -- defines the node names for the diagram, the associated messages and screen control for each node.
- (4) Body statements -- describes the structure of the diagram and its transition conditions.

The body statement contains information about the arcs of the transition diagram. In a transition diagram, several arcs may emanate from a node or subconversation. The arc statement describes selection conditions for arc traversal and may have the following sections:

- 
- (1) on selector — where selector is one of the following:
    - a string expression enclosed between single quotes, (e.g. 'admit').
    - a subconversation name. This allows a variety of inputs for selection of a single arc.
    - a variable name defined in the variable definition section. If the user input matches the variable range conditions, the user input is stored in the variable and that arc is selected.
    - a digit. This case implies that the source of the arc is a subconversation and the digit is a value returned by the subconversation.
    - skip. This option causes the arc to be traversed without waiting for user input. If skip is used, then no other selectors should be given.
    - else. This is the default selector. If the user input does not match any of the above selectors or if the input is null, this arc is selected.
  - (2) to destination — where destination is the name of the destination of the arc, which may be a node name or a subconversation name.
  - (3) do action — where action is an integer number corresponding to an action defined in a user supplied routine.
  - (4) when action\_return — where action\_return is a list of integer values, returned by a user supplied function, that may be used to traverse different arcs dependent on the outcome of that action — where return\_value is an integer value which is returned to the calling conversation. This facility permits error conditions to be transmitted and permits the invoking conversation to proceed along different paths depending on events in the subconversation.

A typical combination is to associate an action and a destination with a selection. In that way, specific input causes an action to be performed and places the system in a new state awaiting the next input. That case is shown in the first example, while the second example shows the use of the when and return constructs.

Examples:

```
arc CAD on '4' do 5 to DL_4
      on '5', 'HELP' do 99 to HELP_NODE
      on '3' to NEXT
      else do 29 to LAST
```

```
arc NODE_H on 'DONE' DO 5 when 5,6,7 to NODEZ
      when 1,2 to DONE return 4
```

User action routines then provide needed system functions. They may be supplied by the user in the form of C, Fortran, Pascal or PLAIN functions. User routines are supplied through an integer function named "actions(acnum)" where "acnum" is an integer, which may be used in a case statement (computed goto in Fortran). Unix system calls, including the invocation of shell scripts, may be included in actions by using the routine "system(command)" where "command" is a valid Unix command. The action function should return a positive integer value that corresponds to the return values expected in the RAPID code following the "when action\_return" section of the arc statement.

---

The following example could appear in an action file name "actions.c"

```
actions(acnum)
int acnum;
{
    switch (acnum) {
    case 1:
        char *var;
        var = getvar("varname","convname");
        return(1);
    case 2:
        to_troll("scriptname",paramcount,param1,param2);
        chgvar ("varname","convname",newval);
        return(1);
    case 4:
        if (user_function(arg1,arg2))
            /* user_function could be in another file */
            return(2);
        else
            return(10);
    default:
        return(0);
    }
}
```

The actions provide a way to link the script to the Troll/USE relational database management system. The RAPID/USE system contains several procedures and functions to simplify the linkage.

The procedure to\_troll calls a script from a designated Troll/USE library and passes parameters to the script. The function getvar permits a RAPID variable to be passed as a string to the action routines. The procedure chgvar allows an assignment to be made to a RAPID variable from the action routines, so that the new value may be displayed. Other procedures simplify initialization, error handling, and termination for the database system. These features are especially useful since most interactive information systems provide conversational access to data.

Hence, the transition diagrams of RAPID, combined with the Troll/USE relational database management system, is particularly useful for building functioning interactive information systems. We have successfully used RAPID to create interfaces and prototypes for several interactive information systems. One such system is an interactive program to aid in the specification, design, and evolution of Troll relational databases. Work is continuing on additional systems, which we intend to include with subsequent versions of the User Software Engineering tool distribution.

#### References

- A.I. Wasserman, "The User Software Engineering Distribution: an Overview," in *Information System Design Methodologies: a Comparative Review*, ed. T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart. Amsterdam: North Holland, 1982.
- A.I. Wasserman and D.T. Shewmake, "Automating the Development and Evolution of User Dialogue in an Interactive Information System," in *Evolutionary Information Systems*, ed. J. Hawgood. Amsterdam: North Holland, 1982.
- A.I. Wasserman and S.K. Stinson, "A Specification Method for Interactive Information Systems," *Proc. IEEE Computer Society Conference -- Specifications of Reliable Software*, Cambridge, MA, 1979.

Chairperson: *Berkley Tague*  
Bell Laboratories

## The UNIX System: New Directions

*Berkley A. Tague*

UNIX Development Laboratory  
Bell Laboratories  
Murray Hill, NJ

This talk reviewed the history of UNIX and discussed the highlights of and new directions being taken with System V.

The ancestry of the C language was traced from CPL (Strachey) to BCPL (Richards) to B (Thompson) to C (Ritchie). The ancestry of the UNIX operating system showed BESYS (Bell Labs) and CTSS (MIT) influencing Multics (Bell Labs, GE, MIT) and Multics and Genie (UC Berkeley) influencing UNIX Versions 1–5 (Thompson). The Version 6 to Version 7 transition was centered around portability research while the V7 – 32V – System III transition was oriented toward improving program development tools.

Bell uses UNIX for a wide variety of applications including program development, document preparation, telephone network operations support, laboratory control, and manufacturing support. In phone support use UNIX is usually used with some data base system. In laboratory and manufacturing use they typically use satellite processors to handle data interrupts.

System III (and V) came from the integration by the UNIX System Developers of the systems developed by the research group (V6, V7, and 32V) and those developed by the Applications Developers (PWB 1.0). The offering of System V is the first time the current working Bell version of UNIX has been offered. In the past the version offered was typically two years behind that being used inside Bell.

A list of the features of System V was presented: file system enhancements, performance improvements, improved interprocess communications, many quality improvements (about 1600 fixes and enhancements), library improvements, *uucp* improvements, *ex* and *vi* editors, accounting enhancements (raw data collection), line printer spooling system, improved operator interface, new peripheral support, and improved documentation. [Most of these topics were covered in some detail later in the session.]

New directions for UNIX within Bell were listed as:

- control of system evolution through a system definition, the ANSI C standard, and the /usr/group and European UNIX Users Group standards efforts;
- meeting market needs of unbundling products and “small” UNIX systems;
- providing compatibility between versions;
- retaining portability and simplicity; and
- continued development and support.

In response to a question Mr. Tague stated that the research group that developed V6, V7, and 32V will not be offering any more complete UNIX systems.

---

# UNIX File System Evolution

*Larry A. Wehr*

UNIX Development Laboratory  
Bell Laboratories  
Murray Hill, NJ

This talk enumerated the file system improvements made between Version 7 and System V. The objectives in this development were to maintain upward compatibility, increase performance and reliability, and to maintain a simple and uniform environment.

The file capacity has changed from 1mb files and 32mb file systems with V6 to 1gb files and 8gb file systems with V7 and System III to 16gb files and file systems with System V (with 1kb blocks). System V has a new compatible file type with 1kb blocks for improved performance. A flag has been defined near the end of the superblock to indicate the file system type. System V can have (different) file systems with both 512 byte and 1kb blocks existing on the same system; the utilities will handle either type. It can handle files from previous versions back to V7.

A number of changes have been made for file system reliability including ordered writes for *create/link/unlink*. There are also several performance enhancements besides the new 1kb block file type. These include an expanded range for the number of system buffers (up to 1000), a physical I/O buffer pool, and increasing the *stdio* buffer size to 1024 bytes.

Future work on the file system is being done in the areas of networking and data base applications, including record / file locking, archiving, and access methods.

## Evolution of UNIX System Performance

*Jerome Feder*

UNIX Development Laboratories  
Bell Laboratories  
Murray Hill, NJ

This talk told of the changes in System V that make it run 25% faster than System III on a VAX 11/780 on their timesharing benchmark. The same benchmark showed very little performance change when run on a PDP 11/70. Very little was said of the 11/750; it appeared to be about the same as an 11/70.

The C libraries, and especially *stdio*, have had code refinements, have some better algorithms, use line-at-a-time output to terminals, and have had some routines recoded in assembly language.

Many changes have been made in the kernel. System call overhead and context switch are around 25% faster on a VAX. Inode search time is now constant on a VAX because a hashed search is used. (The 11/70 still uses a linear search.) Pipe bandwidth is approximately unchanged for 512 bytes per transfer but is about 2/3 higher for 1kb per transfer (VAX only). Disk I/O is reduced with the larger (1kb) buffers and the larger number of buffers available.

Raw and normal terminal I/O run at the same speed on a VAX with the programmed KMC/DZ! However, terminal I/O on a normal DZ on a VAX runs slower than on an 11/70 with a DH11.

Bell's timesharing benchmark measures system throughput for a standard set of commands under increasing load. It is modeled on the Bell Labs program development environment. There is no networking or system administration and user actions are read from files, not terminals. There are no swaps.

---

Several questions were asked:

Assembly language in the C library: every function that can be expressed in C is; for some there are also assembly language versions.

Demand paging: System V does not do it on a VAX and they will not commit as to when it might be implemented.

System V vs. 4.1: 4.1 is about 12% faster than System III and 12-13% slower than System V on their benchmark.

## C Programming Environment

*D. J. Kretsch*

UNIX Development Laboratory  
Bell Laboratories  
Murray Hill, NJ

This talk discussed a proposed standard C environment based on System V and talked about software generation systems used with Bell Labs. The standards effort is aimed at increasing the portability of C code and making it more usable on small systems.

The proposed standard C environment has two levels: level one with core features that would be required for all C environments, and level two with additional UNIX-specific features. Level 1 would be a proper subset of level 2. Local features could be added to either level. The standard would cover the language (based on the Kernighan and Ritchie book), C library, *stdio*, the math library, and the C preprocessor.

[It was unclear to me how this standard would be formalized: through the ANSI C standard committee? Or would it be just a Bell definition? ...Ed.]

Mr. Kretsch also described the Software Generation System (SGS), a set of compilation and object file manipulation tools for developing software for the host UNIX environment or other computers. The tools are divided into machine dependent and machine independent pieces. The tools include a C compiler, assembly language optimizer, assembler, linkage editor, disassembler, various object file manipulation routines, and a library of object file access routines.

SGS runs on many host processors, and it is easy to change the target processor. The underlying feature that allows SGS to be portable is a common object file format. This is consistent across all processors and it allows such things as the definition of memory directives for the linkage editor.



---

# UNIX System Definitions and Standards

*Don Cragun*

UNIX Development Laboratory  
Bell Laboratories  
Murray Hill, NJ

This talk described ongoing work within Bell to define a full and subset (Bell) UNIX system. The subset — level 1 — would be for small systems. Level 2 would be a full UNIX system — System V is an example.

At each level the definition would include hardware, operating system calls and drivers, and utilities and libraries. Each level would allow for the addition of packages including hardware to replace or add-on features.

The standards effort is based on System V. They are working with the /usr/group Standards Committee. The standard is to cover all level 1 system calls, selected level 2 system calls, and the level 2 C environment. Part of the intent of the effort is to control the evolution of UNIX and to provide UNIX releases with upward compatibility “wherever possible” and “advance notice of non-compatible changes.” [But see Mr. Guffy’s comments on compatibility in session W3B...Ed.]

## Questions and Answers

[Just the answers were recorded; hopefully the questions can be inferred.]

Dr. Bob Fabry of UC Berkeley made several comments: Bell is aggressively importing good ideas developed outside the Labs [like the larger buffer]; support of UNIX by Bell as well as DEC and other vendors is important.

Not much work has been done on argument specification compatibility.

SGS is on the VAX only and not all the pieces are there.

There have been some improvements in real-time capabilities; they are interested in hearing what improvements are needed.

The symbolic debugger is called *sdb*; it works at the C statement level.

The question of defining whether things like *nraff* and *yacc* should be part of level 2 is being looked into.

System V handles both the old and new *a.out* formats; files in the old format need to be converted only if they are going to be processed by the link editor.

SGS is used at Bell for cross compilation to the 8086, 68000, and Western’s 32-bit processor. It has not been decided whether they will offer a 68000 or 8086 version of SGS.

System V has not been run on the PDP-11/44; Bell decided long ago to ignore it.

Chairperson: *Keith Muller*

## **A Uniform and Simple User Interface to UNIX**

*Spencer Rugaber*

Interactive Systems Corporation  
Estes Park, CO 80517

This presentation described an interface to UNIX which has been under development at Interactive Systems Corporation's Estes Park, Colorado facility. Their stated goal was to produce software that would be self-teaching and easy to use by non programmers. Their results, a full-screen interface that front-ends UNIX and a product upon which an entire office automation system has been based.

ISC's solution was a program which runs on top of UNIX that utilizes a full-screen editor with a function key-pad to execute some basic operations. They have tried to limit the number of commands the user must confront, as well as bring continuity in command usage across the fields of usage, i.e., mail, word processing or database retrieval.

After running a prototype system for the last two years at their Colorado facility, they claim that someone could begin working with this new interface within a half-hour using the self-help that is included within the system.

---

## A Uniform and Simple User Interface to UNIX\*

~~~~~

Spencer Rugaber  
Interactive Systems Corp.  
October 25, 1982

The UNIX operating system is currently available on many machines and may be expected to spread even more widely in the near future. As it spreads, it will be used by people with many different backgrounds. Since UNIX was designed by programmers for programmers, it is not surprising that many first-time users are taken aback by its terse and esoteric commands and its dearth of error messages. Because it has grown by accretion, UNIX also contains many disparate and non-uniform pieces. Consequently, a user faced with learning a new segment often derives little benefit from what he or she has already learned. For example, ed, egrep, sh, expr, cpio, lex, and sed each have their own slightly different pattern matching notation. Another drawback is that UNIX was devised for hardcopy teletype devices. It is notable, for example, that the Bell terminal driver has no page-holding mode and that no screen editor is released with UNIX.

In order to take advantage of UNIX's strengths: ubiquity, portability, and abundance of applications, it is desirable to overcome its weaknesses in the user interface area. In particular, it is desirable to devise a user interface to take advantage of the bandwidth provided by a terminal screen. The interface should be as uniform as possible; that is, there should be a small number of operations that are used by all applications. Moreover, the interface should be self-teaching and easy to learn by people without programming backgrounds.

This paper describes the steps that we have taken to devise such an interface, the problems that we have encountered, and our approaches to solution. The result is a full-screen interface that acts as a front-end to UNIX and upon which an entire office automation system has been based. The same operations that one uses to manipulate files (deletion, copying, renaming) are used for word processing, electronic communications, and data base retrieval. The underlying model is that of a screen editor acting on a hierarchical data base.

The UNIX file system can be considered as a hierarchial (tree-structured) data base, with text files as leaves on the tree. Many applications, however, use data that is structured (has internal organization). Uniformity is gained if the internal structure of such files is naturally appended to that of the files system. Thus the commands for moving the point of reference in the file system can be the same as those for moving around a structured file. Similarly, the commands for deleting a line in a text file,

---

deleting a record in a structured file, or deleting a file from a directory are identical.

The user interface to UNIX uses the full screen. The cursor can be moved to any line before effecting an operation, just as in a screen editor. Further, it is straightforward for a user to design the screen layout in such a way that the necessary data is readily accessible. This subsumes many of the functions of data entry systems and data base retrieval mechanisms.

The interface consists of about ten basic operations. These include moving the cursor on the screen, scrolling the data, zooming in to see more detail and out to see a summary, copying and deleting data, and formatting. In addition to these, it has been found desirable to allow each application to define a set of special functions. For example, an electronic mail system not only needs to edit text but also needs to send a message, forward it, or file it in a file drawer.

Naive users need on-line help facilities and a self-teaching capability. Furthermore, they need to feel safe from making mistakes. Our help facility allows the user to request information about any operation. Moreover, it is designed so that it is easy for an application builder to add information about a new application.

It is natural for a system that is based both on a hierarchical data base and on a screen editor to provide a self-teaching capability consisting of files that the user can edit as he or she learns. Details are hidden by the hierarchical structure until the user is ready to confront them.

In order to feel safe, users must be sure that alterations that they make to the data base are recoverable. Currently we have three levels of safety. First of all, any basic destructive operation is reversible. That is, the RESTORE operation recovers a deleted line, file, or mail message. If the user discovers at some later time, after the session is over, that an alteration should be undone, he or she can play back the entire history of changes to a file and select the appropriate moment for reversion. Finally, basic operations that destroy files automatically back them up.

This system has been running in prototype for two years. It has proved to be easy to learn by non-programmers. It has also proved to be a fertile medium. Current applications include electronic mail, phone messages, calendars, reminders, accounting, timesheets, trip reporting, inventory, word processing, simple programming, and many others. Many of the business functions of the company for which I work have been adapted to this interface, which encourages internal consistency. Perhaps the ultimate proof of its effectiveness is that people enjoy using it.

---

\* UNIX is a trademark of Bell Telephone Laboratories, Inc.

---

# UNIX Time-sharing Menu-driven Office System for Terminals (UTMOST)

*James A. Neyer*

Perkin-Elmer  
Computer Systems Division  
106 Apple Street  
Tinton Falls, NJ 07724

This presentation covered the enhancements made by a group at Perkin-Elmer to UTMOST which is a menu-driven office system developed by the Air Force Data Services Center.

Initially, they started this project by converting it from V6 to V7 UNIX, and then proceeded with “demilitarizing” to make it more suitable for commercial applications.

The present version has eight sub-systems: help, word processing, document management, electronic mail, electronic bulletin board, (these last two based on the Rand interface, though another could be used) an executive support package for things like do-lists, appointment schedules, a system information sections, and finally a facility for extending the system with your own personal menu or hierarchy of menus, without recompilation of source programs.

This package is available now and they have been looking for beta-sites. It will also be made available for the next software distribution tape from USENIX.

## A Friendly Text Processing Environment

*Arthur Zemon*

Software Productivity Project  
TRW R2/2009  
One Space Park Drive  
Redondo Beach, CA 90278

AUTHOR is a program developed as part of an ongoing software productivity project internal to TRW.

The user interface of the editor is similar to the Berkeley *vi* editor which takes AUTHOR based text and displays *nroff*d like output on the user's screen.

Their approach was to put together something which looks like a dedicated word processor running under UNIX. AUTHOR's interface supplants the complexities associated with *nroff*/*troff* with a set of commands which the user must learn and some special terminal keys for formatting functions like paragraph, section headings, etc.

The text which the user sees approximates what one would get after it had been processed by *nroff*. An *nroff*/*troff* input file is produced by AUTHOR. A positive plus is that AUTHOR still allows raw *n*/*troff* formatting commands to be inserted into the AUTHOR file. If AUTHOR doesn't understand these, it leaves them as is for processing later by *n*/*troff*.

There are no commercial marketing plans at present for AUTHOR, though AUTHOR's authors will proudly show it off if you are interested.

---

# Writing User Documentation for UNIX Systems

*Jean Yates and Rebecca Thomas*

Yates Ventures  
P.O. Box 22411  
San Francisco, CA 94122

A quick and lively rundown of Yates Ventures which covered a brief review of the recent marketing survey, along with short, verbal vignettes of the new UNIX user who is *not* the computer scientist/data processing professional of today's UNIX.

They have several books forthcoming that are targeted for the business-oriented micro user, and programming oriented text targeted for the commercial applications programmer who only wants to know how to make the system work for their purposes and needs.

In a nutshell, "a system manual is just not enough for the new breed of end users" and Yates Ventures is actively filling this void.

---

**This page intentionally left blank**

---

Thursday, 27 Jan 1983

Session: T2B — New UNIX Implementations I

Chairperson: *David Patterson*  
Institution missing

## Hewlett-Packard's Entry into the UNIX Community

*Frederick W. Clegg*

Technical Computer Group Engineering  
Hewlett-Packard Company  
11000 Wolfe Road  
Cupertino, California 95014

After more than two years in preparation, Hewlett-Packard Company has made a major commitment to the UNIX operating system. The first public unveiling of HP's support for this burgeoning industry standard occurred in November 1982 in conjunction with the announcement of the powerful new HP 9000 series of 32-bit computers.

The HP 9000 employs a new high-density, high-speed NMOS technology to achieve performance levels heretofore unapproached in a desktop workstation. The CPU chip alone of the HP 9000, for example, contains nearly a half-million transistors. The 32-bit system architecture provides features to support high-performance virtual memory, networking, and multiple CPUs. The resulting workstation is targeted primarily at scientific and computer-aided-engineering applications.

Shortly after the introduction of the HP 9000, HP will announce its support for UNIX on its Series 200 desktop line (two members of which are the popular HP 9826 and HP 9836 computers). These machines employ the widely-used MC68000 processor in a compact and cost-effective package aimed at instrument system control, "computer-aided-work," and other applications not requiring the expandable performance of the HP 9000.

The HP 9000 and the Series 200 both represent major new computer family lines from Hewlett-Packard. A rapid progression of successors is anticipated. UNIX is to play a central role in both of these families, as well as in other HP computer product lines in the future.

This paper will sketch the history of UNIX at HP. The major system characteristics of both the HP 9000 and Series 200 systems will be summarized. The UNIX implementation approaches of both families will be presented, and a glimpse of the role of UNIX elsewhere in HP's product line will be provided, including a sketch of HP's plans for some significant and unique extensions to industry-standard UNIX capabilities.



---

HEWLETT-PACKARD'S ENTRY  
INTO THE UNIX COMMUNITY

Frederick W. Clegg  
Section Manager

Hewlett-Packard Company  
Technical Computer Group Engineering  
11000 Wolfe Road  
Cupertino, California 95014  
...!ucbvax!hpda!fc

Hewlett-Packard Company has made a major commitment to the UNIX\* operating system. The first public unveiling of HP's support for this burgeoning industry standard occurred in November 1982 with the announcement of the powerful new HP 9000 series of 32-bit computers.

The HP 9000 employs a new high-density, high-speed NMOS technology to achieve performance levels heretofore unapproached in a desktop workstation. The resulting workstation is targeted primarily at scientific and computer-aided-engineering applications; it is nonetheless a general-purpose computer well-suited for any other application area requiring high performance.

HP will shortly announce its support for UNIX on its Series 200 desktop line (three members of which are the popular HP 9816, 9826, and 9836 computers). These machines incorporate the widely-used MC68000 processor in compact and cost-effective packages aimed at instrument system control, technical office automation, and other applications not requiring the expandable performance of the HP 9000.

The HP 9000 and the Series 200 both represent major new computer family lines from Hewlett-Packard. UNIX is to play a central role in both of these families, as well as in other HP computer product lines in the future.

This paper will sketch the history of UNIX at HP. The major system characteristics of both the HP 9000 and Series 200 systems will be summarized. The UNIX implementation approaches of both families will be presented, and a glimpse of the role of UNIX elsewhere in HP's product line will be provided, including a sketch of HP's plans for some significant and unique extensions to industry-standard UNIX capabilities.

---

\*UNIX is a Trademark of Bell Laboratories.

## 1. HISTORICAL PERSPECTIVE

In October, 1979, a small team was assembled within Hewlett-Packard to explore possible directions for future technical and scientific workstations. Early on in this investigation, it was determined that UNIX was emerging as the de facto standard operating system throughout much of the computer science and engineering community. Later, the word length and performance of the then-current 16-bit processors in use were judged to be of questionable adequacy for the proposed user base and interest shifted to a new VLSI 32-bit processor being built by HP's Desktop Computer Division (DCD). Corporate interest in UNIX, however, remained undiminished.

Originally, the VLSI "superchip" processor was envisioned as providing the heart for a new BASIC-only super-desktop-computer which would provide an upward growth path from the popular HP 9845. As the full potential of the 450,000 transistor superchip and its supporting "NMOS-III" semiconductor technology became more apparent, objectives were expanded to embrace virtual memory, multiple languages, multiple users, and other attributes not previously associated with HP "desktops". UNIX was seen to provide an attractive vehicle to achieve these additional goals, while the burgeoning popularity of UNIX as an industry-wide operating system standard became increasingly indisputable. Initially cooperative efforts toward implementing UNIX on the superchip were pursued within HP's Technical Computer Group in California and at DCD in Colorado. In January 1982, full responsibility for UNIX on the superchip was given to DCD's Engineering Systems Operation, which would soon be spun off as the new, separate Engineering Systems Division with the charter to propagate the superchip-based family. Such was HP's commitment to this new computer family.

Meanwhile, interest in UNIX had spread within HP across a spectrum of divisions producing computing products ranging from small personal machines to large mainframes. To facilitate communication, standardization, and leverage of implementation work across product lines, a corporate UNIX Steering Committee was formed in April of 1981. This body continues to meet monthly to further the original objectives. In addition to the Steering Committee itself, several subcommittees have been commissioned as well, including the "Technical Working Group", "Networks Support Group", "Marketing Working Group", etc.

In November of 1982, the superchip made its first public product appearance with the announcement of the HP 9000. At the same time HP announced its support of "HP-UX", HP's implementation of UNIX -- with extensions.

In the same time frame, the developers of HP's "Series 200" desktop computers (currently including the HP 9816, 9826, and 9836) committed themselves to support UNIX on at least some of the products in that family. The first "Series

210" machines, as these UNIX-running products are to be called, will be announced in summer of 1983. Plans to support UNIX in a variety of other product lines are being actively pursued, as well, and indeed UNIX has become a major central theme in much of HP's computer business.

At this point, some definition of nomenclature is in order to avoid possible confusion in the following discussions. Strictly speaking, the term "HP 9000" denotes the entire set of new-generation HP desktops, including both the superchip- and MC68000-powered models. The formal designation of the former are HP 9000, Series 500, 600, and 700, in its single, double, and triple CPU configurations, respectively; the HP 9816, 9826, and 9836 (and eventual follow-on models) are collectively designated the HP 9000, Series 200 machines. UNIX systems in this latter, MC68000-based, family will be called the "Series 210", to distinguish its members from earlier BASIC, PASCAL, and HPL models. For brevity, however, in the balance of this paper we will use what has become popular HP parlance by using the term "HP 9000" to refer exclusively to the superchip-based members of the Series 500-700. We will usually use simply Series 210 to collectively refer to the MC68000-based machines.

The balance of this paper will explore the HP 9000 and its implementation of HP-UX in some detail and compare these with the forthcoming Series 210 family. We will conclude by summarizing HP's strategic plans for evolving beyond current UNIX technology to become a leader in this important marketplace.

## 2. THE HP 9000

The first Hewlett-Packard product to offer HP-UX is the HP 9000, which was publicly announced on 16 November 1982. HP-UX is one of two different operating system offerings available on this product, the other one being a single-user environment built around a version of HP BASIC extended to embrace graphics, database management, and the system's other capabilities.

The HP 9000 is targeted principally to improve the productivity of technical professionals. The major intended application areas are CAD (computer-aided design) and CAE (computer-aided engineering). There are currently applications packages to support electrical and mechanical engineers. Its System-III-compatible UNIX services provide enhanced programmer productivity in a widely known and acclaimed way.

### 2.1. HP 9000 Hardware

The HP 9000 is available in several packaging and performance options. At the heart of all of these is the so called "superchip" set. These VLSI devices are the first to incorporate HP's new "NMOS-III" technology and are, by a substantial margin, the highest density integrated circuits in commercial use in the world. Typical devices employing this technology have 1 micron geometries and run at 18 MHz clock rates. The superchip CPU contains 450,000 devices. The device count of the memory chips of the family is even higher.

The density and speed characteristics of the superchip set would exceed the power dissipation capabilities of conventional packaging technologies suitable for an office environment. For this reason, and also to increase circuit board densities achievable, HP developed the "finstrate". This is a copper plate with a Teflon\* dielectric layer which is used in lieu of conventional printed circuit boards. The high heat conductivity of the copper allows this medium to serve both as a heat dissipation "fin" and as an interconnecting "substrate" for the chips -- hence the term finstrate.

Three major module types make up a typical HP 9000 system: a CPU finstrate, an I/O processor finstrate, and a memory finstrate (current capacity: 256 Kbytes). These finstrates are interconnected by the "MPB", the 32-bit, 36 megabyte/second memory-processor bus comprising the system's spinal cord. Higher performance configurations employ multiple finstrates of each type on the MPB, up to a total of twelve finstrates. The HP 9000 was expressly designed to support multiple CPU systems, with processing power nearly proportional to number of CPUs installed.

---

\*Teflon is a registered trademark of E.I. duPont de Nemours and Co.

The HP 9000 CPU has a stack-oriented architecture, with over 200 instructions in the instruction set. Virtual memory is implemented with a paged segmentation scheme. While the largest single segment is limited to 512 Kbytes, multiple segments can be grouped in contiguous segment table entries to provide, in effect, a single paged data object of up to 512 Mbytes.

## 2.2. HP 9000 HP-UX Implementation

As is well known, there are basically three levels of software in a UNIX system: commands, system intrinsics, and function libraries (corresponding to sections 1, 2, and 3 of the familiar "UNIX Programmer's Manual", respectively). The HP 9000 implementation of HP-UX has achieved the first and third of these levels largely by "porting" licensed C source code from AT&T's System-III and other origins (such as the bsd releases from U.C. Berkeley). The kernel (which supports the section 2 services) is a different story on the HP 9000, however. The HP-UX kernel is implemented in a layered fashion atop an HP-proprietary kernel called "SUN". SUN is implemented in MODCAL, an enhanced version of PASCAL. MODCAL supports information hiding via a module concept similar to MODULA's, an error recovery mechanism, and systems programming extensions such as absolute addressing.

Several motivations prompted this somewhat unconventional implementation of the HP-UX kernel on the HP 9000. It should be understood that much of the low-level, machine-dependent software for the HP 9000 was complete before a final commitment was made to offer UNIX on this product line. Thus reinvention of device and interface drivers, low-level memory management routines, power-up code, and architecturally-dependent utility routines was avoided by this approach. Additionally, this approach permitted leverage of software developed to support graphics and database management from the BASIC environment.

Additionally, SUN has a number of features which are not present in AT&T UNIX. These provide opportunities for HP-UX to make a contribution above and beyond other UNIX implementations. Such features include real-time performance in the area of interrupt response time and process switching, support for multiple processors, and reliability in the face of system errors. Also HP's highly acclaimed IMAGE database management system was already largely implemented in MODCAL atop SUN and the approach taken greatly eased the task of making IMAGE available to HP 9000 users opting for the HP-UX environment.

Such a "layered" kernel implementation does involve some risks. These were identified early in the investigation phase of this project. Firstly, the HP 9000's kernel comprises all new code, introducing significant risk of functional incompatibility with System-III and other industry-standard UNIX implementations. Thus extensive validation efforts were undertaken to ensure that full compatibility be achieved. Secondly, tracking the evolution of UNIX standards as new releases appear from AT&T, Berkeley, and elsewhere will involve more effort than mere "porting" of new source code. Additionally, the need to

perform conversions between SUN and UNIX formats does increase operating system overhead somewhat.

The better approach to use was so far from obvious, that HP (in a tactic very unusual for this company) pursued BOTH paths with parallel teams for over a year. In the end, however, HP management decided that the attendant advantages outweighed the risks in opting for the layered kernel used by HP-UX on the HP 9000.

Although most commands and library functions on the HP 9000 were "ported" from licensed source code, as noted earlier, significant extensions beyond AT&T's System III are also present in HP 9000 HP-UX. The most widely recognizable of these include

- \* A virtual memory implementation which provides both variable size virtual segments and fixed size virtual pages
- \* HP PASCAL
- \* HP's IMAGE Database Management System
- \* HP's AGP-3 three-dimensional and DGL two-dimensional graphics subroutine packages
- \* Ethernet-compatible 10 Mbit/sec local area network support
- \* U.C. Berkeley's vi screen editor

Additional information on the HP 9000 and its implementation of HP-UX may be found in references [1 - 6].

### 3. HP's SERIES 200 DESKTOPS

In February 1982, Hewlett-Packard announced the HP 9826 desktop computer. This machine represented the first member of a new family of machines which has been designated at the "Series 200". In its initial offering, the HP 9826 was promoted primarily as a descendant of the earlier HP 9825, available in BASIC, HPL, and PASCAL versions. (HPL is a terse, HP-proprietary language similar in flavor to that employed by its popular handheld calculators.) With its extensive support for HP-IB (i.e. IEEE 488) peripherals, it is naturally well-suited to automated instrument system control; thus it was principally targeted at this marketplace in the beginning.

The first HP desktop computer to employ the Motorola MC68000 processor, the HP 9826 possessed ample performance to address numerous other applications as well. Especially with the advent of a second member of the family, the HP 9836 with a considerably larger display and dual minifloppy disc drives, the Series 200 quickly found its way into numerous other application areas. Visicalc (R) and several project management applications packages were soon added, as well. The HP 9836 quickly found favor with HP's crafters of in-house VLSI CAD tools. The family rapidly developed a reputation as a workhorse for what, in its diversity, came to be called "CAW" (computer-aided work -- i.e. text and document processing, miscellaneous arithmetic computations, electronic mail, spreadsheet calculations, other project planning activities, etc.). In November, the repackaged, small "footprint" HP 9816 was added to the series, which is now well on its way to becoming HP's most popular line of desktop computers. Indeed, within two months of its introduction, the 9816 had become HP's fastest-selling computer product ever!

To add excitement and momentum to this new family, a commitment was made in mid-1982 to add a full UNIX offering to this family as well. It is anticipated that the efforts now well underway will produce publicly available products later this year. These variants will constitute what will be called the "Series 210".

#### 3.1. Series 200/210 Hardware

As already observed, the machines of this family employ the popular Motorola MC68000 processor. Current models use the 8 MHz part and have a 625 nsec memory cycle time. Hardware memory mapping is to be provided in the HP-UX versions using a scheme which imposes less than 10% overhead on memory access compared to a "raw" 68000. Upgrading a current HP 9826 or HP 9836 to run HP-UX is being designed to be a simple and inexpensive process.

Both series are highly modular and offer a rich selection of plug-in interfacing and accessory modules including RS-232, IEEE-488, high-speed parallel and DMA interconnection provisions.

The standard-equipment, bit-mapped graphics capability is further augmented by the availability of optional color displays.

IEEE-754-format floating-point numbers are added, subtracted, multiplied, and divided by firmware using high-performance microcode.

Local area networking is available now using HP's Shared Resource Manager. This capability uses what is physically a star configuration to provide users with what appears to be a bus to a shared file system. One may mix and match HP 9000 Series 500-700 and Series 200/210 machines at will within the same network. Interconnection via industry-standard Ethernet (R)/IEEE-802 links will be available by the end of 1983.

Although the first of the Series 210 line will use the same MC68000 processor as the Series 200, together with an HP-proprietary memory management unit, plans are already well underway to incorporate the already-announced next-generation descendants of the MC68000, including those offering wider word width and virtual memory support, as these parts become available.

### 3.2. Series 210 HP-UX Implementation

Somewhat in contrast to that of the HP 9000 described earlier, the Series 210 HP-UX implementation followed largely the course of a conventional UNIX "port", starting initially with some of the work of the "TRIX" project at M.I.T. and of a software team at HP Labs as "bootstrap" tools. A good deal of the kernel code and most of the libraries and commands from UNIX Release 3.0, as licensed from AT&T, was used intact where this was not precluded by the latter's use of assembly language and other machine dependencies. For performance reasons, a 1 Kbyte block size (adjustable via a system parameter) was adopted for the file system. Confirming the experiences of others, the port was straightforward and even early "breadboard" versions simply "woke up and ran", according to developers.

Although AT&T's System III was used as a defining base, the Series 210 HP-UX implementation provides a steadily growing list of enhancements. Autodial support for selected non-Bell modems has been added to uucp, drivers for many of HP's diverse peripheral lines have been developed, and popular additions from outside sources (such as Berkeley's vi editor) have been ported. Other HP-value-added enhancements, as discussed elsewhere in this paper, are also being provided.

Additional information on the Series 200/210 computers and their HP-UX implementation may be found in References [1], [7], and [8].



#### 4. HP's UNIX STRATEGY

That Hewlett-Packard has made a major corporate commitment to the UNIX operating system is evident. Why did this come about and where is this supposed to lead?

HP has long recognized the tremendous investment represented by operating system software for its computer products. From this realization, some major corporate goals have evolved. Firstly, it was recognized that HP needs superior functionality, together with interfaces which make that functionality conveniently accessible to people, computer programs, and physical processes. Secondly, the need was perceived to provide a rapidly increasing degree of compatibility. This had to include support and growth paths for our existing customers, as well as a standard interface which will allow applications programs written by HP and its customers to run on a variety of HP architectures.

The search for potential starting points for a compatible HP operating system arrived at UNIX as the clear winner. Although various operating systems, including earlier HP-proprietary offerings, outshine UNIX in various specific areas, UNIX provides the best overall operating system with wide industry acceptance. There is every reason to believe that UNIX will become the de facto standard operating system for 16- and 32-bit computers to an extent which will dwarf the phenomenon whereby CP/M\* provided such a standard for 8-bit microcomputers. Furthermore, UNIX is particularly strong as a development environment, which is crucial to our own productivity and that of our customers and our industry.

##### 4.1. HP's Contribution

UNIX's acceptance in the marketplace has stimulated a large number of competitive offerings. What will distinguish HP-UX from these other products? To begin with, HP-UX is not "just UNIX". HP is extending its capabilities into the areas of distributed computer systems, database management, graphics, real-time processing, and others to go far beyond any UNIX thus far defined in the marketplace.

HP is implementing HP-UX over a very wide spectrum of product lines and architectures. To facilitate this, the HP-UX standard consists of the definition of an interface, not a particular body of code. To accommodate differing architectures and capabilities, differing implementation strategies will be appropriate. Modularity in two dimensions is needed. Firstly, HP-UX is modular in its functional capabilities. For example, there is a graphics "slice" of the standard which is irrelevant to machines having no graphic I/O capability, but which ensures that when the same capabilities are present in two different HP

---

\*CP/M is a registered trademark of Digital Research, Inc.

systems their invocation and functioning will be identical. Other slices are provided for real-time, networking, database management, etc.

A second dimension of modularity is demanded by varying machine comprehensiveness. Future implementations of HP-UX range from inexpensive turnkey application engines to sophisticated, high-performance, and often distributed hosts and servers. We have defined five "levels" of comprehensiveness. The top two levels of the HP-UX standard correspond to full HP-extended UNIX and to industry-standard UNIX, respectively. The three lower levels are targeted to provide more limited program development or applications execution environments on configurations constrained to small hard discs, floppy discs, and memory only, respectively.

Modularity between and within slices and levels, carefully designed to achieve these objectives, together with an orderly and tightly controlled evolution of the HP-UX standard, has been endorsed by a firm commitment by Hewlett-Packard's top management. The compatibility to be achieved by adherence to this standard, as ensured by use of a single corporate HP-UX validation suite, will permit leverage of such efforts as the writing of user documentation. HP's desire to be certain of ability smoothly to import to its hardware the vast amounts of anticipated third-party software will further motivate careful tracking of external standards.

Perhaps most important among HP's planned contributions to the UNIX community, however, is the element of support. HP is the largest organization to date to announce full hardware, software, and overall support for UNIX systems. This will include, of course, the after-sale support which make so many of its customers so loyal to HP. Some of the of the areas in which UNIX takes the greatest criticism today, such as learning and reference aids for users and maintainers, are obviously areas where a much higher level of support than has been available in the past should be welcomed by many. HP's experience in producing documentation and operating phone-in consulting services for customers will doubtless play key roles here.

#### 4.2. Future Plans

Much of HP's plans for future UNIX offerings is reflected in the above discussion of the HP-UX standard. The presence of various "slices" reflects our intent to exploit UNIX's advantages both on machines with graphics capabilities as well as those without, for example. The various planned "layers" reflect the fact that HP will seriously examine the desirability and feasibility of offering HP-UX on a very wide spectrum of products conceivably ranging from easily portable self-powered machines to large high-performance mainframes.

The fact that we are expending significant amounts of time and other resources on defining and evolving "the HP-UX standard", holding steering committee and working group meetings, and the like reflect a conviction widely

shared within HP. This conviction is that HP-UX can play a major positive role in providing increasing consistency, uniformity, compatibility, connectivity, and software portability -- both across our product lines, and between our products and "the rest of the world".

We realize, however, that these advantages can only be achieved through carefully planned, negotiated, and documented standards, and subsequent commitment to these standards by all concerned parties. Hewlett-Packard has made a major commitment, through top management levels, to pursue such a standard. We anticipate that, through this commitment, HP will become, alongside Bell Labs, U.C. Berkeley, and others, a leader in determining the future directions of UNIX technology.

## 5. CONCLUSION

Hewlett-Packard Company has made a major corporate commitment to the UNIX operating system. The first implementations of HP-UX have been announced, several more are imminent, and still more are underway for other current and future products. HP-UX will be available as an option on all of HP's technical computers, as well as selected business, personal, and instrument computers.

With HP's proprietary additions to UNIX, implemented uniformly across a broad range of machines, we intend to provide a level of features and cross-system compatibility unmatched in the industry.

## ACKNOWLEDGEMENTS

The author gratefully acknowledges the numerous helpful contributions to this paper and the work it describes from many colleagues in HP's Technical Computer Group, its Desktop Computer Division, and its Engineering Systems Division. Especial thanks are due to Jim Bell and Rich Hammons of TCG; John Bidwell, Jeff Bork, and Donn Terry of DCD; and Jeff Lindberg and Ken Heun of ESD.

#### REFERENCES

- [1] The HP-UX Reference Manual, Hewlett-Packard Co. Part Number 09000-90004 (available Apr 1983).
- [2] HP 9000 Computers, Hewlett-Packard Co. Part Number 5953-4616 (Sep 1982).
- [3] The HP-UX Operating System, Hewlett-Packard Co. Part Number 5953-9402 (Sep 1982).
- [4] Jeff Lindberg, A Layered Implementation of the UNIX Kernel, (to be submitted to "Software Practice and Experience" in Jan 1983).
- [5] HP-UX System Manager's Manual, Hewlett-Packard Co. Part Number 97089-90045 (available Apr 1983).
- [6] HP 9000 Configuration, Information, and Ordering Guide, Hewlett-Packard Co. Part Number 5953-4615 (Oct 1982).
- [7] Series 200 Family Brochure, Hewlett-Packard Co. Part Number 5953-4645 (available Feb 1983).
- [8] Series 200 Family Price List, Hewlett-Packard Co. Part Number 5953-4635.

---

## **4.2BSD on the Sun Workstation** **(or What we Did on our Summer Vacation)** **(or How to Emulate a VAX on a 68000)**

*Tom Lyon and Bill Shannon*

Sun Microsystems, Inc.  
2310 Walsh Avenue  
Santa Clara, California 95051

This talk will describe the porting of 4.2BSD to the Sun Workstation, a 68000-based graphics workstation with high speed local networking capability. We will discuss the architecture of the Sun system and the impact it had on the 4.2BSD port. The true meaning of location zero and its relevance to UNIX will also be described. Performance of the system will be mentioned.

We will also talk about the networking capabilities of the Sun Workstation and the Sun Fileserver. Future plans for the Sun Workstation and the Sun 4.2BSD system will be described.

## **Experiences in Porting 4.1BSD UNIX to the $\lambda$ 750 VLSI Development System**

*Paul Chen and Chet Britten*

Metheus Corporation  
P.O. Box 1049  
Hillsboro, OR 97123

The  $\lambda$ 750 system is a multi-processor based VLSI design workstation. We selected 4.1BSD UNIX to be its operating system. This paper presents the experience we gained in the process of porting UNIX to the  $\lambda$ 750 system.

The  $\lambda$ 750 system architecture presented a unique set of porting problems. In order to achieve high system performance, especially for graphics applications, we distributed the function of the UNIX kernel among three MC68000 processors. This paper describes the ways we partitioned these functions and some techniques we used to enhance performance. A paged memory management scheme is used with a non-mapped user structure. Three-port memory is used for processor communication and data sharing. Window management is elegantly handled and naturally mapped to the existing tty driver interface.

To accomplish the task under tight budget and time constraints, we developed several tools and some integration skills. Some experiences in task partitioning and scheduling are also discussed.

---

# UNIX on Apollo Computers (Yet Another UNIX Emulation)

*Eric R. Shienbrood*

*Carl A. Soeder*

*James R. Ward*

and

*Kincade N. Webb*

Apollo Computer, Inc.  
15 Elizabeth Drive  
Chelmsford, Massachusetts 01824

An Apollo computer node is a high performance, personal workstation with a bit-mapped display. A high speed local area network is used to interconnect multiple nodes.

Its operating system, Aegis, provides demand paged virtual memory, multiple processes, and a transparent network-wide file system.

A system program called the Display Manager manipulates windows on the bit-mapped display, and acts as an input-output medium between the user and the system.

The UNIX shell can be invoked in a window and can execute concurrently with Apollo shells and UNIX shells running in other windows. UNIX commands can be invoked by either the Apollo or UNIX shells.

The talk will discuss topics such as:

- why we chose to emulate rather than port UNIX
- our solutions to interesting implementation problems such as a emulating *fork* on a virtual memory machine
- changes to Aegis and the user environment to accommodate UNIX
- what facilities are not provided

Results on the success of the emulation will be presented. Some general problems of UNIX emulation will be discussed, and comparisons will be made to other UNIX emulations.

---

AUX on Apollo Computers  
(Yet Another UNIX(\*) Emulation)

by

Eric R. Shienbrood, Carl A. Soeder,  
James R. Ward, Kincade N. Webb

Apollo Computer Inc.  
15 Elizabeth Drive  
Chelmsford, MA 01824

## 1. Introduction

The UNIX operating system has proven to be an increasingly popular and comfortable environment for program development and other computing activities. As a result, a large and varied body of software is now available to UNIX users. The user who can run such software has the potential to tap into the results of thousands of man-hours of past and future development efforts. In addition, UNIX provides a familiar environment that assures a user that he can switch from one manufacturer's system to another without having to relearn how to interact with the system. This remains true even though the systems may be running on very different hardware. In response to these considerations, Apollo Computer Inc. has developed AUX, a UNIX System III-compatible software environment.

AUX is more than just a set of routines whose names are identical and whose functions are roughly analogous to the UNIX system calls. Rather, it provides a faithful emulation of the UNIX software and user environments on a personal workstation that supports network and graphics display capabilities. Programs that run under UNIX System III run under AUX with little or no modification. The user who is comfortable interacting with UNIX can make the transition to AUX almost immediately.

## 2. The Apollo Domain System

Each Apollo node is a high performance single-user workstation with a high resolution (800 x 1024 pixels) bit-mapped display and an optional Winchester disk. The display includes a hardware bit-mover, and comes in both color and monochrome varieties; the latter is available in either a landscape or a portrait orientation. Nodes are interconnected via a proprietary high speed (12Mb) token-passing ring network. Aegis, the object-oriented Apollo operating system, provides demand paged virtual memory and a network-wide hierarchical naming space, which is syntactically very similar to the UNIX directory hierarchy. A user (or program) may transparently refer to any object in the network in a uniform way, regardless of where in the network the object resides.

\* UNIX is a trademark of Bell Laboratories

---

The network forms the backbone of the system. It is the medium through which the total universe of objects and resources is available to the user. Within the network, objects are named by 64 bit unique identifiers (UID's). These UID's are unique both in space (among all present and future Apollo nodes) and in time. Since each object comprises a linear 32 bit address space, an entire network forms a global 96 bit address space. Within the operating system, a naming server is responsible for mapping the user-oriented pathnames into UID's, just as in UNIX, the naming server maps pathnames into device/inode pairs. It should be emphasized that it is the large network-wide address space, and the way objects in it are manipulated, rather than any particular CPU instruction set, that characterizes the Apollo system.

A process runs within a 24 bit address space. Fundamentally, the only way for a process to manipulate an object is to map part or all of it into the process's address space. Conversely, every valid part of a process's address space corresponds to some object (whether named or unnamed) in the filesystem. When referenced, the pages of an object are brought in either from a node's local disk or from another node on the network, depending on where the object is found.

Each resource within the system is managed by a piece of code known as a type manager. For example there are type managers that manage mapped objects, serial I/O lines, magnetic tape drives, and interprocess communication mailboxes. Every object in the filesystem has a type UID attached to it, which identifies the type manager that should handle that object. User programs could access resources directly through the operations provided by their type managers. Instead, however, they usually make calls to the Streams Manager. Streams provide a uniform, device independent interface to the system's resources. The Streams layer is responsible for translating calls on its operations into calls on the type manager for the object being manipulated. The operations supported by Streams are very similar in nature to the UNIX system calls for manipulating files and peripherals.

Within each node, a system program called the display manager handles all interaction between programs and the human user, as well as file editing and some system control functions. The display manager allows the human user to view and control multiple independent activities concurrently. It divides the screen up into multiple windows whose size, shape, and placement are under user control. Two types of windows are supported. One provides a view port on an edit pad, displaying all or part of an object in a window, and allowing the user to modify the object if he has the proper access rights. The other type of window displays a portion of a virtual terminal, known as a pad, to which a process writes output and from which it reads keyboard input. The pad contains a complete transcript of everything read and written by the process. Multiple processes may run concurrently, each sending output to its own pad. Typically, each process window initially runs an interactive Apollo shell. The user may extract regions of text from any window and insert the text into an edit pad or process input pad. Finally, user programs can perform graphics operations, either within a window, or by taking over the entire display.



---

Several aspects of the system design were strongly influenced by concepts from UNIX. The most prominent examples of UNIX-inspired ideas are the system's process-rich structure, and the use of the Streams interface as a means of achieving device-independent I/O. In addition, many of the original utilities on the system, including the shell, came from the Software Tools group of programs. This heritage help insure UNIX a comfortable "fit" within the Apollo environment.

### 3. Apollo Software Architecture

As in UNIX, the process is the basic agent of computation. However, the structure of an Apollo process is more complex than in UNIX. For one thing, the operating system nucleus is mapped into each process's address space (protected from access by user programs, of course). Libraries, instead of being linked in with each program, are also mapped into each process's address space. Both the operating system and the libraries are demand paged, like all other objects. The read-only parts of the libraries, including the executable code, are shared among all processes on a node. For this reason, these are called global libraries. The writeable parts of the libraries are allocated freshly for each new process. Only the read-only parts can be initialized to values other than zero.

Many tasks which in UNIX are performed by the kernel have been implemented in global libraries. The Streams layer, for example, resides within a global library, as does the user-level process manager. The nucleus of the operating system contains only that code which must run in a privileged mode.

In a departure from UNIX, the Apollo system allows more than one program to be invoked within a process. This kind of program invocation is very much like a procedure call, and in fact stack frames are allocated on the regular user stack. Whenever a program is invoked in-process, a "level-push" takes place. That is, resource usage levels are recorded before the program runs. After the program finishes execution, a "level-pop" takes place, in which any resources consumed by the invoked program are released. Aside from allowing fast program invocation, this mechanism permits programs to produce side effects which remain in force after program termination. For example, the Apollo command to change the working directory is a program, and is not built into the shell.

Three types of disk files are supported by the Streams manager. These are record structured with fixed length records, record structured with variable length records, and an unstructured (byte-stream) format. When the AUX project began, files were created with record structure by default. In text files, a single line of text typically constituted one record; reading from such files would return data only up to the next newline character. Seeking within record structured files required reading through all the records preceding the seek destination. It was clear that this behavior would not be appropriate for emulating UNIX file characteristics. Furthermore, we realized that the performance of Apollo software would be improved if most files were unstructured. We therefore introduced a new file

---

type called UASC (for unstructured ASCII), and a new Streams call that would return a buffer full of data rather than just a single record. UASC is now the default creation type for files created by software in both the Apollo and AUX environments.

To protect objects, the Apollo filesystem attaches an access control list, or ACL, to each object. Just as UNIX associates user and group identifiers with each process, Apollo maintains person, project, and organization identifiers (PPOs), collectively known as a subject identifier, or SID. An ACL contains one or more entries, each giving the rights to be granted to the specified PPO. Any or all of the three fields of an entry may be replaced with a wildcard, in which case the ACL entry matches no matter what the corresponding component of the process's SID is.

#### 4. Goals

Given this well-developed and highly useable environment, a simple port of UNIX to the Apollo hardware would have implied a significant loss of functionality. Put another way, we felt that a UNIX emulation for Apollo systems would combine the advantages of both systems. To maximize the utility of the product, we had to provide a software environment as close as possible to that of UNIX System III. At the same time, it was extremely important that the Apollo and AUX environments be fully integrated. The user who has never seen an Apollo system should be able to work in the UNIX shell, run UNIX utilities, and in general, "feel" as though he is using a UNIX system. The key sequences he has "stored in his fingers" should produce the expected behavior from the system. This would allow him to begin using the facilities offered by the Apollo Domain at his own pace to increase his productivity. He should be able to freely intermix Apollo and AUX programs, even within a single pipeline, without any awareness of the programs' origins.

We began this project with two advantages. First, the Apollo system offered, for the most part, a superset of the functionality provided by UNIX. Second we were free to modify the system software as necessary, within reasonable limits. On the negative side, it was clear that there would be certain aspects of UNIX that we could not emulate very well, if at all. Among the reasons for this were:

- o There was no Apollo mechanism that was even roughly equivalent to the UNIX feature in question.
- o Supporting the UNIX feature would have created unacceptable difficulties in the Apollo environment.
- o The benefits of supporting the feature were simply not worth the cost.

In addition, there were examples of UNIX functionality which would have been expensive or impossible to support, but for which there was equivalent Apollo functionality in a different form.

---

## 5. Implementation Overview

The code that implements the emulation resides in a global library. This library actually comprises two layers. The inner layer is the code that implements the "UNIX virtual machine", that is, the system calls. The outer layer contains the routines that make up the C library -- standard I/O, string routines, and so on. This layer's presence in a global library allows single-module C programs to be compiled and run without going through a link stage.

There is one problem associated with in-process program execution in the presence of global libraries. Since there is only one copy of writeable data per process, all program levels in a process see the effect of any changes made to that data. The solution is to make the allocation of writeable data part of the level push/pop mechanism. When some variable is first needed on a new level, storage for it is dynamically allocated. All of this is, of course, completely invisible to user programs.

The following paragraphs touch briefly on a few specific aspects of the AUX implementation.

### Exec

In emulating the exec system call, we decided that it should replace only the current program level within a process, rather than replacing the entire process image. Exec, then, entails a level-pop followed immediately by a level-push, as if the old program had exited and the new program was then invoked. This allows a process to be running an AUX shell, which calls an AUX program, which then execs another program. When the exec'd program finishes, it returns to the shell, as one would expect.

Normally, the exit call terminates only the current program level. If exit is called in a forked process however, the entire process is terminated. This properly handles the case of a process that runs an Apollo shell, which then invokes an AUX shell, which then forks and execs a program. If the child process were not terminated, the exec'd program would return to the unsuspecting Apollo shell in the child process. Meanwhile, the AUX shell in the parent process would still be waiting for the child to terminate.

### Memory Allocation

Some UNIX programs assume that successive calls to sbrk return contiguous pieces of memory. The Apollo mechanism for dynamic memory allocation is not guaranteed to behave this way. Therefore, the first time a program calls sbrk, a single large chunk is set aside. Sbrk requests by that program are then doled out from this chunk until it is exhausted. The default size of the pre-allocated chunk is one-quarter megabyte. However, a new call provided in AUX allows a program with a large appetite for memory to explicitly set this size.

### Protection

---

When a C program running in the AUX environment creates a file, an ACL is set up that matches the mode specified in the creat call. The ACL contains four entries:

- o The first gives the person identifier of the process creating the file, with the rest of the fields left as wildcards. This entry specifies the "owner" rights for the file.
- o The second gives the project identifier of the process, with the rest of the fields left as wildcards. This specifies the "group owner" rights for the file.
- o The third contains wildcards in all fields, and specifies the rights granted to "others".
- o The last gives the "backup" project read access to the file. This allows tape backups of the object to be made.

The chmod call modifies the ACL to reflect the new permissions.

### Filenames

The Apollo naming server allows letters, digits, underscores, dollar signs, colons, and periods to be used in filenames. It does not distinguish between upper and lower case. UNIX, of course, allows any character except slash and null to appear in a filename. For AUX, we decided to compromise by making case significant and adding a handful of new special characters. This is accomplished in the AUX library by mapping pathnames presented by C programs into character sequences acceptable to the Apollo naming server. The inverse mapping is performed when a directory is being read by a C program.

Both Apollo programs and users running in the Apollo environment (i.e., shell) expect that case is insignificant in filenames. For this reason, the mapping just described only takes place for C programs that use the library calls like open, unlink, and chmod. Even for C programs, the mapping takes place only in the "AUX environment", which is entered by a setting a per-program-level switch. This switch is set by the AUX shell and cleared by the Apollo shell. Its value is inherited if it is not explicitly set or cleared. The same switch, by the way, determines whether the mapping of UNIX modes to ACLs, described earlier, takes place.

### Reading Directories

A new type manager was added to Streams to allow directories to be read as files. However, one problem remains. In UNIX, the number of entries in a directory can be determined if the size of the directory file is known. In the Apollo system, this is not true, since a directory is not simply a linear array of entries. Furthermore, the only way to determine the number of entries is to scan through the entire directory. That would make the stat call unacceptably slow. This discrepancy affected two of the standard UNIX commands.

---

## 6. Variations from UNIX

In weighing the costs and benefits of supporting each UNIX feature, we tried whenever possible to justify implementation. For those features that we did emulate, our goal was to provide behavior identical to that in UNIX. Thus for example, our unlink call does not cause a file to be deleted until the last process holding the file open closes it. As another example, we mimic the behavior of UNIX with regard to which system calls are interrupted by caught signals. On the other hand, there are a number of areas for which we provided only partial support, or even no support at all. Most of these are features that are not commonly used by UNIX programs, or else are very easy to live without, so that their omission detracts very little from the usefulness of AUX. The following list describes how we departed from a full implementation of UNIX primitives:

- o No support for the link system call. Although Apollo filesystem objects can be catalogued under more than one name, Apollo programs observe a policy of not doing so. The reason for this is that each object contains a pointer to its parent directory. This enables us to determine the full pathname for any object. If the name under which an object was originally catalogued were removed, we would no longer be able to backtrack up the naming tree to construct the name.

Instead of UNIX-style hard links, Apollo supports soft, or symbolic links. A symbolic link, rather than pointing directly to an object, points to a pathname. When the naming server encounters a symbolic link, it automatically inserts the specified pathname in place of the link component, and then continues its search. In many cases, the behavior of soft links is to be preferred, since a soft link can span physical volumes, and can never refer to an out of date version of an object. At the call level, we found that no more than half a dozen UNIX programs used the link system call. Those uses all fell into one of two categories. First, link was often used just prior to an unlink call in order to rename a file. Second, link was sometimes used in the implementation of a lock routine. For the first use, we have added a rename routine, and for the second, a lock routine that is callable by user programs. In addition, we have supplied soft\_link and soft\_unlink calls.

- o No support for setuid or setgid programs. At present, Apollo has no mechanism for implementing protected subsystems. When we do, setuid support will be added. Also, reflecting the single-user nature of the system, we do not have any concept analogous to the super-user, nor is there any distinction between real and effective user and group id's. Generally speaking, those programs that are normally setuid to root under UNIX are allowed to run without special access rights on our system. Along the same lines, we provide no support for the setuid and setgid system calls, since we presently have no mechanism for altering the SID of the user once he has logged in.
- o No support for the chown system call. Since the Apollo filesystem does not really support the concept of the owner of a file, we did not believe that this call was critically important.

- 
- o No `/dev/mem` or `/dev/kmem`. This rules out the use of programs that try to read system data structures.
  - o Limited support for `ioctl`. Most of the UNIX terminal modes map into capabilities provided by the Apollo serial I/O line driver. For virtual terminals, however, `ioctl` has no effect.
  - o The semantics of `creat` differ in the following way from those of UNIX: In UNIX, the file descriptor returned by `creat` always allows both reading and writing the new file. The file descriptor returned by our `creat` call grants only the permissions specified in the mode argument to the call.
  - o `Mknod` works only for fifos (named pipes). It does not allow creation of block and character special devices. Apollo mechanisms are available for creating objects of various types.
  - o No support for the `ptrace` system call, which is used only by debuggers. We have our own source-level debugger, and so had no need to support the UNIX debuggers.
  - o No `profil` system call. Apollo has other facilities for profiling both user programs and the operating system.
  - o Other UNIX system calls that are not supported are `sync`, `mount`, `unmount`, `acct`, and `stime`.

Some of the omissions in our emulation can be characterized as areas in which UNIX violates principles of abstraction and information hiding. For example, the `who` command in its original form could not run because the Apollo system does not maintain the file `/etc/utmp`. If there were library calls which gave information about who is logged on, and if these calls were the only means for obtaining such information, we could have supplied those routines and supported commands such as `who` without modification. Instead we had to entirely recode the program to call directly upon Apollo services. Another example of this problem is the way that programs obtain information about processes by reading kernel tables in `/dev/kmem`, rather than by making system calls.

## 7. Results

One way to evaluate the success of the emulation is to consider how much effort was required to get the standard UNIX commands running. Of the programs distributed with UNIX System III, about 115 were supported in our first release of AUX. These include most of the large programs, such as the shell, `nroff`, `tbl`, `eqn`, `make`, `yacc`, `lex`, `lint`, `awk`, `ed`, `sed`, and `diff`. Since then, we have added others, including `uucp`, `vi`, and the pre-job control version of the C-shell. About two-thirds of the programs required no modification of the source code.

---

Some of the changes we made corrected true bugs. The most common was the tendency of some programs to carelessly reference through nil pointers. In an Apollo process, location zero is not readable by user programs. Other programs had hardwired 14's where they dealt with the size of a directory entry. An Apollo pathname component may have up to 32 characters. A few programs depended on automatic variables having an initial value of zero. Still other contained machine-dependent code that would have failed on any 68000-based system. Very few of our changes are attributable to a failure to mimic UNIX semantics. Two problems already mentioned - one concerning creat and the other concerning directory reading - forced a modification of about a dozen lines of code in three programs. Those few programs that used the link system call were recoded to use rename, soft link, or lock.

Some changes were made for the sake of efficiency. In programs that need to determine the current working directory, we replaced invocations of the pwd program with calls to a much faster library routine. The shell was modified to invoke most programs in-process instead of always forking. Forking now occurs only for programs that run in the background, programs that are part of a pipeline, and shell files.

## 8. Conclusion

The project leading to the initial release of AUX required approximately one and a half man-years of effort, exclusive of the work on the Apollo C compiler. Future work will include performance tuning, with particular attention to improving the performance of the fork primitive. We will look at other versions of UNIX, including Berkeley's, to determine to what extent we can support the unique kernel features they offer. We will also continue to look for interesting UNIX applications programs, particularly those that will prove useful in both the AUX and Apollo environments.

We have found that a carefully designed emulation can be quite faithful to the semantics of the target system. The best results are achieved by following general principles of good software design. The emulation should be regarded as a set of transfer functions or "protocol converters" that map between the viewpoints of the two systems. Therefore, the emulation should maintain as little of its own state as possible, relying instead on well-advertised services offered by the native system. Finally, the stronger the conceptual similarities between the native and emulated systems, the more successful the emulation will be.

## References

Nelson, D., Apollo Domain Architecture, Apollo Computer Inc., 1981.

Dolotta, T.A., et al, UNIX User's Manual, Release 3.0, Bell Telephone Laboratories, 1980.

Chairperson: *Bill Appelbe*  
University of California, San Diego

## Towards a UNIX System Ada Programming Support Environment

*J. Eli Lamb*

Bell Laboratories  
Murray Hill, NJ

Ada<sup>†</sup> is a Department of Defense-defined programming language that was defined for portability. It is suppose to run in an Ada Programming Support Environment (APSE), which was defined in the Stoneman requirements document. The APSE provides lifecycle support for Ada software development and consists of a portability interface, environment database, basic Ada toolset, and project-specific tools. The APSE interfaces to the host system through a virtual operating system Kernel APSE (KAPSE) and database.

Many of the APSE requirements are similar in notion to UNIX — simple and consistent interface, tool concept, powerful command interpreter. The database APSE requirements are significantly beyond what is normally found on UNIX systems.

UNIX System V meets many of the APSE requirements. Significant extensions will be required to meet the database requirements.

## LINUS (Leading Into Noticeable UNIX Security)

*Steven M. Kramer*

Mitre Corporation  
Bedford, MA 01730

This talk described types of security problems on UNIX and two levels of solutions.

The major security problems with UNIX were listed:

- global (unrestricted) privileges like those available with *su*;
- authentication and simple passwords;
- "trojan horses" — utilities that look like normal commands but that perform other or additional things;
- integrity and denial of services (e.g., printing 10<sup>6</sup> bytes);
- badly designed system programs;
- leniency in file mode settings; and
- lack of mandatory security levels.

Linus III is an add-on, portable solution to these problems that is built on existing UNIX security features and requires no kernel modifications. It tries to limit superuser use as much as possible, allows privileged command auditing, and does file system integrity checking. Linus III has been implemented but is not packaged. An accompanying database to provide additional controls has been designed but has not been implemented.

Linus IV is a more extensive package that is based on 4.1BSD. It adds Department of Defense security levels to UNIX as well as handling the other problems mentioned above more completely than with Linus III. It requires kernel modifications as different security levels must be in different hardware domains. Changes must also be made to about six programs.

<sup>†</sup> Ada is a trademark of the Ada Joint Program Office — US Government.



---

**This page intentionally left blank**

---

## UNIX Logo

*Brian Harvey*

Atari, Inc. and SESAME Group, UC Berkeley

Atari, Inc.

1196 Borregas Avenue

P.O. Box 427

Sunnyvale, CA 94086

This talk gave a brief introduction to the Logo programming language and its use in pre-college education. Mr. Harvey also announced the availability of Logo on UNIX and that he has submitted an updated version of it to USENIX for the 1983 Software Distribution Tape<sup>\*</sup>.

Logo was designed for learning programming and for learning in general. The ideas came from artificial intelligence research and LISP and Piaget's research into how children develop thinking skills. It is procedural, interactive (interpreted) and recursive. It has list processing and is not typed<sup>#</sup>.

Part of Logo is the notion of *turtle graphics*, a simple and simply-expressed method of doing graphics. A turtle on the screen has a relative position and direction. It can be rotated or moved and can leave a trail of its movement (i.e., draw a line).

---

<sup>\*</sup> available to 1983 Institutional members of USENIX with any UNIX license

<sup>#</sup> There is a informative article on Logo by Mr. Harvey on page 163 of the August, 1982, issue of *Byte*.

---

## Unix Logo

Brian Harvey  
Corporate Research  
Atari, Inc.  
1196 Borregas Ave.  
PO Box 427  
Sunnyvale, CA 94086

### Abstract

Logo is a programming language designed for use in education. This paper presents an overview of what Logo is, the current status and future plans for the Unix implementation, a note on the implementation of turtle graphics for a number of different graphics devices, and some remarks on the use of Unix in pre-college education.

### Introduction to Logo

Logo represents the coming-together of two sets of ideas: from computer science, the ideas of artificial intelligence, and from education, the developmental psychology of Jean Piaget. This synthesis was the contribution of Seymour Papert, along with colleagues at Bolt, Beranek and Newman, Inc., and later at MIT.

From Piaget comes the idea that intellectual development depends on the learning of certain central, powerful ideas which help in organizing many pieces of knowledge, and that these ideas are learned not so much by direct teaching as by immersion in a culture in which the ideas are used. The design of Logo tries to emphasize such powerful ideas as debugging, procedures, locality, and recursion. The context in which these ideas are used is designed to fit as closely as possible with the ordinary life of the people using Logo.

The specific application areas are largely the contribution of artificial intelligence. The central idea is the metaphor of the computer as symbolic information processor; specific AI applications from natural language to robotics have proven to be intuitive and interesting to students. As a programming language, Logo is closely related to LISP; in fact, it could be described with some justice as LISP disguised as BASIC.

These days, many people associate Logo with the idea of turtle graphics: a technique of computer graphics in which lines are drawn relative to the computer's immediate, local point of view rather than in terms of an absolute Cartesian coordinate system. Although turtle graphics did originate in Logo, it is only one part of the Logo world, and historically rather a late development. A turtle graphics package is not Logo; other vital parts of the language include recursive procedures and the list processing capability to support the linguistic data hierarchy of letters, words, and sentences.

---

## The Unix Logo implementation

The current version of Logo is based on a 1979 version written by Douglas B. Klunder for The Children's Museum in Boston. That original version, although written in C and YACC, ran under RSX-11M, not Unix. It was based on a very early Logo specification from BBN, at a time when words and sentences had been invented but not yet full list processing. The Klunder implementation also had no graphics support.

I converted the original version to run under Unix at the Lincoln-Sudbury Regional High School in 1980. I also added list processing primitives, and support for the Terrapin floor turtle, a small computer-controlled robot. This work was done under version 7 Unix on a PDP-11/70. The following year, the school got some Atari 800 personal computers to use as graphics terminals, supporting turtle graphics. A version of Logo at about this stage of development, hereafter called release 1, was on the 1981 Usenix distribution tape.

The addition of list processing to the Klunder implementation strained the capabilities of the storage allocation mechanism in that version. The early Lincoln-Sudbury system, although quite featureful, tended to blow up every so often because of bugs in the storage allocator. Most of these problems were fixed during 1981 by one of my students, Jay Fenlason, then a high school sophomore. He also added many missing primitives to the language.

In 1982 I reorganized much of the interpreter in order to make it run faster. (It used to be terrible, and is now merely bad.) The new version, release 2, was distributed directly to only a few sites. It was, however, the basis for the current version, release 3, which will be distributed through Usenix.

With a few exceptions, release 3 is source language compatible with Apple Logo and with the forthcoming Atari Logo. It is already in use in several installations, so it has gone through a lot of debugging. A major new feature in release 3 is the PAUSE command, allowing interactive commands in the local context of a procedure for dynamic debugging. Turtle graphics is supported on the Atari 800, the DEC GIGI, the ADM-3 with the Retrographics board, the Tektronix 4014, and the Sun workstation.

The Children's Museum, which holds the copyright on the original version of this Logo interpreter, has approved its unlimited free release. It may not be sold. Nobody is offering guaranteed software support for this Logo, but I am interested in getting bug reports and have so far been able to provide fixes for the bugs I've heard about. I would particularly like to thank Don Martin and his students at the College of Marin for heavily exercising this version of Logo. Finally, I should make it clear that despite my employment at Atari, this Unix Logo is completely unrelated to the forthcoming Logo interpreter for the Atari home computers.

---

## Future plans for Unix Logo

There is a great deal which could be done to improve this Logo interpreter. In the area of execution speed, the current release now spends most of its time in the innards of YACC, and I don't think I can do much to improve it without another major reorganization, for which I am too busy. I would be delighted for someone to take over that project, or even rewrite Logo from scratch.

I do plan a release 4, with two major changes. One is to make it lint. The current release generates about 25 blocks of lint errors, down from about 75 blocks in the first release but still not good enough. The second major change is in workspace management; I plan to implement packages using Unix subdirectories as the organizing structure for procedures. I'm not promising when this will be done, since it has to be squeezed between my work at Atari and my full-time graduate studenthood at the Group in Science and Math Education at UC Berkeley.

Meanwhile, back at Lincoln-Sudbury, Jay, who is now a senior, is working on a Logo compiler. It will compile Logo procedures into C code, so it will be portable to any Unix system. It should make it possible for the first time to write real production programs in Logo.

## The turtle graphics implementation

The first graphics terminal supported by this Logo system was the Atari 800 computer. To make this work, I had to write a graphics terminal program for the Atari. This program is a modification of the Chameleon terminal program written by John H. Palevich. What makes all this interesting is that the Atari "player-missile" graphics hardware makes it possible for the Atari itself to worry about drawing the turtle, so the host computer doesn't have to draw and erase it explicitly. This makes the graphics programming interface more complicated than a single vector-drawing routine for each terminal type. At the opposite end of the range of terminal hardware, the Tektronix 4014 with its storage-tube display can't erase vectors without erasing the entire screen, so Logo doesn't display a visible turtle on it at all.

The solution used in Unix Logo is to define a display terminal type in terms of the structure shown in figure 1. This structure is not presented here as a model of elegance, but the reverse: the moral of the story is the incredible number of kludges required just to support the five kinds of terminals Logo currently knows about.

As mentioned above, displaying the turtle is the first problem. The Atari does it all by itself. The Tektronix can't do it at all. The others do it by drawing and erasing vectors, except that the Sun and the GIGI can use the preferred exclusive-or technique, while the ADM-3 must explicitly draw and

explicitly erase. For all these reasons, a procedure to draw the turtle must be provided in addition to the general-purpose drawing procedure.

Color is the next problem. The GIGI simply has eight colors. The Atari has a color map; four colors can be on the screen at once, out of a palette of 256 colors. Unix Logo has primitives to set the color map (used only for the Atari) and to select a pen color. The other terminals are black and white. A slight complication for the Atari is that it has several graphics modes, two of which are used by Logo: the highest resolution black-and-white mode, and a slightly lower resolution, four-color mode. For the Atari, there are two versions of this struct display, one for each mode.

-----

```
struct display {
    NUMBER turtx,turty,turth;      /* current pos, heading */
    NUMBER xlow,xhigh,ylow,yhigh; /* screen boundaries */
    NUMBER stdscrunch;             /* normal aspect ratio */
    int cleared;                   /* nonzero after
                                   first use */
    char *init,*finish;            /* printed to enable,
                                   disable graphics */
    char *totext;                  /* printed for
                                   temporary text mode */
    char *clear;                   /* printed to clear
                                   screen */
    int (*drawturt)();              /* draw turtle, takes
                                   one arg, 0 to show,
                                   1 to erase */
    int (*drawfrom)(), (*drawto)(); /* two args each, x and y
                                   call both to draw
                                   vector */
    int (*txtchk)();               /* called for all graphics
                                   to give error if
                                   terminal is in text
                                   mode and will mess
                                   up the screen */
    int (*infn)(), (*outfn)();     /* called to enable,
                                   disable gfx */
    int (*turnturt)();             /* no args, called when
                                   turtle changes
                                   heading */
    int (*penc)(), (*setc)();      /* color map routines */
    int (*state)();                /* one arg, a char which
                                   indicates some
                                   change of state */
};
```

Figure 1

---

Some terminals can be set up for graphics simply by sending them a character string; for others, something more complicated is required. It was therefore convenient to provide both a character string and a procedure for initialization and for resetting the terminal on exit.

Since the Atari draws the turtle itself, it needs to be told when the turtle turns. There is a special procedure for this purpose. A similar single-terminal kludge is for the GIGI, which doesn't really know whether it is in text mode or graphics mode, so an attempt to draw vectors in text mode will mess up the screen instead of just failing. Therefore, there is a routine just to check for this situation.

Eventually, I decided to combine several small kludges into one routine for each terminal, named "state". This routine is called in several situations, distinguished by a flag character used as its argument. For each terminal, the routine contains a large switch statement in which most cases are ignored.

### Unix in pre-college education

When we got the PDP-11/70 at Lincoln-Sudbury, DEC was very eager for us to use RSTS, their timesharing system for the 11, and not Unix. High schools generally take DEC's advice: while 90% of all U.S. colleges have Unix, I know of no other U.S. high school running Unix. (I've heard some rumors, but none confirmed yet. There is a high school in England running Unix.) Some of the reasons for Unix's popularity among colleges and for RSTS's popularity among high schools are uninteresting: the free academic licenses for colleges, and the fact that people in high schools generally aren't computer experts and don't know better than to do what DEC says.

There are, though, some more interesting reasons. The strengths of Unix, as we all know, are the software tools philosophy, and the easy modifiability of the software. The strengths of RSTS, though, are in administrative control and the availability of Computer Assisted Instruction curriculum materials and software. People who set up computer facilities in high schools generally model their efforts after the Computer Science departments of colleges: they have in mind a particular curriculum through which all students must pass. I feel strongly that high school computer facilities should instead be modeled after the college Computer Center: a service center providing a range of tools, to which the user brings his or her own agenda. It is the latter model to which Unix addresses itself.

Computer professionals familiar with Unix can help by making the power of Unix known to teachers in their local schools. The advent of cheap 16-bit micros with Unix will help too. But it would also be a big help if AT&T would reverse their decision no longer to offer educational licenses to high schools. The system at Lincoln-Sudbury, which got a source license just in time, has been heavily modified both by me and by students. Other schools should have that opportunity also.

---

# A Global Optimizing C Compiler

*George Powers*

Zilog, Inc.  
MS A2-5  
1015 Dell Avenue  
Campbell, CA 95008

Zilog has implemented a global optimizer for C on their System 8000, which uses a Z8001 microprocessor. The optimizer works for either segmented (23 bit addressing) or nonsegmented (16 bit addressing) mode.

The compiler development was done on a descendent of the portable C compiler. Optimization is done in an optional pass after parsing but before code generation. The parser symbol table had to be retained until the code generation pass so the optimizer would have access to it.



---

## A Global Optimizing C Compiler

George Powers  
Martha Laschkewitsch

Zilog, Inc.

27 January 1983

At Zilog Inc., the first practical global optimizing compiler for C is being implemented on the System 8000. It generates code consistent with System III C specifications for the Z8001 microprocessor in either segmented or nonsegmented mode.

The optimizing C compiler is based on Johnson's portable C compiler. Global optimization is done in an optional pass which operates on the intermediate language. The optimizer converts the parser's output from prefix binary trees to quadruples, massages these, and reconverts them to binary trees for input to the code generator. This technique permits meaningful optimization without undue violence to the original compiler's design.

Techniques used include jump optimization, basic block optimization, loop optimization, and global register allocation. Algorithms were derived from recent papers. The major modification needed for most of these algorithms was the inclusion of aliasing information on pointer variables.

The effectiveness of optimization varies widely from one program to another. Some trivial benchmarks run five times faster while other programs cannot be improved at all. The median improvement in a representative set of CPU-intensive benchmarks is 20%.

---

## 1. Introduction

Originally just an adjunct to the UNIX[1] operating system, C[2] has very rapidly become an important general-purpose systems programming language in a wide variety of operating environments. This rapid acceptance is due largely to C's inherent efficiency. In applications where formerly only assembly language was practical, programmers can now use a structured, portable high-level language. C is more efficient than other languages because it more directly utilizes machine resources, with data types like pointer, register storage declaration, and operators like increment/decrement and shift.

Consequently, C is now being widely used in large, time-critical applications like network protocols and operating systems (not just UNIX). While C has proven effective in such applications, any performance improvement is still welcome. A better C compiler could mean higher data throughput in a network, or less CPU demand by an OS kernel. Where applications like DBMS and word processing are coded in C, a better compiler could mean 12 users on a system where only 8 were practical. In short, C is no longer just a research or instructional language, and any effort spent in optimizing its performance is paid back many times in an applications environment.

If C permits such efficient coding, what work is left for a global optimizer? For one, consider the two equivalent functions in figure 1. This function is the bottleneck in a popular benchmark program written by Forest Baskett. Recoding fit() with pointer variables, and removing invariant computations from the loop body, doubles the function's efficiency. Unfortunately, this revision at least doubles its obscurity too; the hand-optimized code is much harder to read and maintain. A global optimizer can perform effectively the same optimizations without modifying the source code.

Figure 2 shows another application for optimization. This code cannot be made more efficient at the source level; the greater part of execution time is spent in function calling overhead, which is beyond the programmer's control. By passing parameters in registers instead of on the stack, the current System 8000 C compiler reduced execution time by 25%[3]. Global register allocation methods make further improvements possible.

---

[1]UNIX is a Trademark of Bell Laboratories.

[2]B.W. Kernighan, and D.M. Ritchie, The C Programming Language, Prentice-Hall Inc., 1978.

[3]Kareemi, Nazim, "Passing Parameters without Memory Bottlenecks", Systems & Software, November, 1982.

---

```

fit (i,j) /* original code */
register int i,j;
{
    register int k;
    for (k = 0; k <= piecemax[i]; k++)
        if (p[i][k])
            if (puzzle[j+k])
                return (false);
    return (true);
}

fastfit (i,j) /* hand-optimized code */
register int i,j;
{
    register int k, n, *pp, *pzp;
    n = piecemax[i];
    pp = p[i];
    pzp = &(puzzle[j]);
    for (k = 0; k <= n; k++)
        if (pp[k])
            if (pzp[k])
                return (false);
    return (true);
}

```

Figure 1. The crux of Forest Baskett's puzzle program.

---

```

main ( ) /* Ackermann's function */
{
    int x;
    x = ack (3, 8);
}

ack (m, n)
int m, n;
{
    if (m == 0) return (n + 1);
    else if (n == 0) return ack (m - 1, 1);
    else return ack (m - 1, ack (m, n - 1));
}

```

Figure 2. A common benchmark of function calling efficiency.

---

---

## 2. The System 8000 Environment

The global optimizing compiler was developed on Zilog's System 8000, also it's primary application environment. The System 8000 is a high-performance supermicro based on the Z8001 CPU and designed specifically to run the UNIX operating system. The Z8001 is a modern microprocessor design with regular architecture and features such as short instruction formats and base-index addressing which promote efficient code generation. Its large general-purpose register file (twice the size of the PDP-11) is a major factor in the design of the optimizing compiler.

One unique feature of the Z8001 among 16-bit microprocessors is its concurrent support of both nonsegmented and segmented addressing modes. In segmented mode, all addresses are 23 bits long, providing a basic address space of 8 megabytes. As in the 68000 architecture, these addresses are typically loaded and stored in two operations on a 16-bit data bus. In nonsegmented mode, program code manipulates only a 16-bit offset, and the CPU outputs a constant segment number stored in a special segment register. This mode, more similar to the 8086 architecture, provides a basic address space of 64 kilobytes. Most programs can be run nonsegmented, thereby avoiding the size and speed penalties of 32-bit addressing.

Zilog's System 8000 UNIX implementation uses three Z8010 Memory Management Units to support both segmented and nonsegmented mode programs. In either mode, programs may have combined or separate instruction/data spaces. System 8000 compilers generate either segmented or nonsegmented code. Where differences exist, this paper will discuss the nonsegmented case. Code generation for segmented mode is very similar to nonsegmented mode, except that addresses occupy a 32-bit register pair instead of a single register.

In System 8000 C programs, function entry is direct and stack growth is performed automatically. The Z8001 performs all four basic math operations on 32-bit operands, and floating-point instructions are emulated by the OS kernel. Therefore, essentially no run-time support for C is required.

## 3. Design Goals

Obviously, the main purpose in creating an optimizing compiler is to achieve higher performance in the object code, which generally occurs at the expense of compilation efficiency. The single greatest constraint on the optimizing compiler's design is its charter as the System 8000's standard production C compiler. As such, it must satisfy a number of conflicting user needs. For instance, during software development, compilation speed is more important than object code efficiency.

---

Another issue affecting design was the schedule and expense of development. Zilog's optimizing C compiler is decidedly not a vehicle for research. Therefore, proven methods of optimization were selected. The optimizing compiler was based on an existing compiler, so that development effort could be focused on the problem of optimization.

Quite commonly, optimizing compilers offer "unsafe levels of optimization." This daredevil philosophy is not well attuned to current attitudes about software reliability and testing. All levels of optimization in Zilog's global optimizer are designed to generate correct code in all cases.

#### 4. Global Optimizer Implementation

The global optimizing C compiler is best understood as a descendent of Johnson's portable C compiler[4] (pcc). Pcc was first adapted to the Z8000 in 1980, by a straightforward port which included a peephole optimizer and generated nonsegmented code. In 1981, this compiler was enhanced in accordance with calling conventions[5] chosen for the System 8000. (This parameter-passing method drastically reduces the overhead in function calls, and permits the intermixing of program modules written in any of the compiled languages on the System 8000.) Also in 1981, the compiler was enabled to generate segmented object code. Since then, it has been released in several revisions on the System 8000, and served as the base for development of the global optimizing C compiler. For the sake of brevity, the System 8000 C compiler before global optimization will be referred to as "pcc."

In order to minimize the costs and risks of optimizing compiler development, changes to pcc were made only where necessary. Optimization is implemented as an optional pass which operates transparently on the intermediate language (IL), between the parser and the code generator passes. When global optimization is not needed, the optimizer is bypassed and compilation is as rapid as usual. Isolation of the optimization process in this manner also facilitates reliability testing.

The greatest changes to pcc were required in the code generator, and next the parser. The peephole optimizer is retained essentially unchanged, but when global optimization is invoked, most of its functions are preempted.

---

[4]Johnson, S.C., "A Portable Compiler: Theory and Practice", Conference Record of the 5th Annual Symposium on the Principles of Programming Languages, January, 1978.

[5]Calling Conventions for the Z8000 Microprocessor, Software Interface Specification, Zilog publication number 03-0130-01, February, 1982.

---

The global optimizer pass is an entirely new addition to pcc. It incorporates long, complex algorithms which require simultaneous access to several very large data structures. Therefore, although the rest of the compiler still is nonsegmented, the global optimizer runs in segmented mode. Implementation in a smaller address space such as the PDP-11's would be nearly impossible.

## 5. Changes to Parser

The parser was changed only as needed to support optimization. Therefore the global optimizer must handle all aspects of optimization, including tasks such as loop recognition, which might otherwise have been assisted by the parser. By avoiding such language-specific dependencies in the IL, the global optimizer can be more easily integrated with compilers for other languages.

In pcc, the parser did all storage allocation, and passed memory offsets for variables to the code generator. However, the global optimizer needs more information to perform significant optimizations. Therefore the new parser passes a symbol table to the optimizer and code generator. Instead of memory offsets, the IL now contains symbol table indices. In addition to the usual information about local and global symbols, the symbol table contains a "reference table" for each function which the global symbols to which the function actually refers.

The IL defined by pcc was not consistent enough for most optimization techniques. In addition to expression trees, it contained actual assembly language code for subroutine prologs and epilogs, switch statements, labels, and jumps. The global optimizer cannot easily make sense of such code. Therefore the optimizing compiler's IL does not permit any assembly language copy lines. New binary expression tree operators are defined for switch statements and function declarations. Branches and labels are expressed symbolically, in new line types.

The parser still generates assembly language data definitions, but these are broken out from the IL and passed along in a separate file. The code generator merges this data definitions file with its output to produce object modules. The global optimizer makes only trivial reference to some floating point constants in the data definitions.

## 6. Tour through the Global Optimizer

The global optimizer pass works primarily on the IL; its output format is a superset of the input format described above. Extensions to the IL will be discussed later, in the context of the code generator. The optimizer uses the symbol table file only for reference purposes.

---

The scope of optimization at this stage of development is intra-procedural. Accordingly, the optimizer works on the code for one function at a time, with reference to local and globally declared data objects. The process of optimization can be broken down into a number of phases which occur in linear order, each building on the results of the last. Figure 3 shows some of these phases and the global data structures created by each phase. The order of optimizations is significant. In general, each phase of optimization requires the data produced by all prior phases.

### 6.1. Quadruples -- Meat for the Algorithms

Prefix binary expression trees, as output by the parser, are not suitable material for familiar optimization methods. Therefore the first thing the global optimizer does to a function is translate its IL representation to quadruples, which are kept in memory throughout the course of optimization. The quadruples are stored in linked list form, in dynamically allocated memory. Most optimizations are expressed as changes to the quadruple list. When the optimizer is finished with a function, the quadruples are translated back into binary expression trees for input to the code generator. This re-conversion is performed only to avoid re-designing the code generator, which uses a template matching algorithm on the expression trees.

Before translating the IL for a function into quadruples, the optimizer reads into memory the symbol table information for that function. It is able to determine which symbol table entries are really necessary by looking at the function's

---

|      | <u>Phase</u>               | <u>Global Data Structure</u> |
|------|----------------------------|------------------------------|
| I    | IL -> Quadruple Conversion | Quadruple List               |
| II   | Basic Block Analysis       | Basic Block List             |
| III  | Flow Analysis              | Predecessor/Successor Table  |
| IV   | Jump Optimization          |                              |
| V    | Alias Analysis             | Alias Matrix                 |
| VI   | Basic Block Optimization   | Kill/Gen Matrices            |
| VII  | Data Flow Analysis         | In/Out Matrices              |
| VIII | Loop Analysis              | Loop List                    |
| IX   | Loop Optimization          |                              |
| X    | Register Allocation        |                              |
| XI   | Quadruple -> IL Conversion |                              |

Figure 3. An overview of the global optimizer pass.

---

---

reference table, thus saving memory space. IL references to tags in the symbol table file are translated into more direct indices to the in-memory symbol table.

Quadruples come in two basic types: those which represent executable code, and pseudo-codes which communicate information to the optimizer and code generator. Typical operation codes for executable code are the basic arithmetic operators, array indexing, jump on condition, call, etc. Typical operands are constants, temporaries, labels, and symbols. Pseudo-op codes will be discussed below, with optimization techniques.

## 6.2. Flow Analysis and Simple Optimizations

The first step of control flow analysis is the definition of basic blocks of code in the quadruple list. A basic block (not to be confused with C language statement blocks) is a sequence of consecutive statements which are entered only at the beginning, containing no branches except possibly at the end. Basic blocks are determined by a simple algorithm[6] which scans the quadruple list from the function entry, marking a block's end at branch statements and following these to the headers for other blocks.

Basic block definitions are represented in a simple list of pointers to pseudo-operators in the quadruple list. These quadruples are inserted at the head of each basic block, and contain the block number, as well as data fields for use in subsequent analysis.

Once the basic blocks are defined, the global optimizer can construct a flow graph for the function. For each basic block, two linked lists are created: one of the basic blocks which might precede this block during program execution, and another of the blocks which might next be executed. These discrete lists are all lumped together in a predecessor/successor table. The header quadruple for each basic block contains two indices into this table: one to the predecessor list and one to the successor list.

The flow graph now provides the information needed to identify dead code. If a block has no predecessors, it can never be executed, and may be safely deleted. Next the global optimizer does jump optimization. Jumps to jumps are simplified into jumps to the final target. Using the flow graph, the optimizer can then determine whether the intermediate jumps have been made unreachable, and eliminate them if that is the case.

---

[6]Aho, A.V., and Ullman, J.D., Principles of Compiler Design, Reading, MA: Addison-Wesley, 1979.



---

### 6.3. Alias Analysis

The existence of pointer variables in the C language, and the freedom of their usage, creates significant problems for optimization. The correctness of many optimizations depends on recognizing when a variable is used, and when it is modified. When pointer variables are present, this task is non-trivial. Different names may in fact refer to the same variable -- optimization algorithms must know where these "aliases" occur.

Consider the program fragments in figure 4. A naive algorithm for optimization might not recognize that an assignment to \*ip could change a value in the array A. Pursuing this course, it seems logical to assign the value A[i]++ (taken before assignment to \*ip) directly to x, since this value might be rapidly accessible in a register. However, when i equals n, the "optimized" code does not assign the same value to x as did the original code.

One response to the aliasing problem is to assume that any value might change, or any value might be referred to, in the presence of a pointer variable. However, pointer variables is so widespread in C programs that with these assumptions, no useful optimizations would be done. Therefore the global optimizer must make a more precise analysis of aliases, to narrow down their scope. The technique used is a modified form of the aliasing scheme described by Weihl[7]. This method does not utilize flow graph information, so that an alias established at any point in a function is effective throughout the function. The scope of

---

#### Original Source Code

```
ip = &A[n];  
.  
.  
A[i]++;  
*ip = 0;  
x = A[i];
```

#### Wrongly Optimized Equivalent Source Code

```
ip = &A[n];  
.  
.  
x = ++A[i];  
*ip = 0;
```

Figure 4. The C language aliasing problem.

---

---

[7]William E. Weihl, "Interprocedural Data Flow Analysis in the Presence of Pointers, Procedural Variables, and Label Variables", Conference Record of the Seventh Annual ACM Conference of the Principles of Programming Languages, (July, 1980), pp. 83-94.

---

alias analysis in the current global optimizer is intra-procedural; optimization algorithms assume that a function call might use or modify any global variables.

Once computed, the alias relations are stored in a bit matrix which indicates the objects to which any pointer might possibly refer. The memory space required by this bit matrix is roughly proportional to the square of the number of pointer variables. In order to conserve space, variables are grouped into several categories. All globals are in a single group, while each structure member is allocated another group. Local variables permit the most productive analysis, and therefore are regarded individually.

#### 6.4. Basic Block Optimizations

With alias information, the global optimizer is now prepared to perform some significant optimizations, not found before in C compilers. Actually the global optimizer works on extended basic blocks, since basic blocks (as defined above) in C programs are generally too short to be worth the trouble. An extended basic block is simply any sequence of quadruples which have a single entry point like basic blocks, but potentially many exits. The algorithm used is a value numbering scheme[8], modified mainly to take aliasing into account. Optimizations performed are common subexpression elimination and compile-time constant folding. Some special-case optimizations also occur in this phase, such as elimination of addition/subtraction by zero and multiplication/division by one.

#### 6.5. Data Flow Analysis and Loop Detection

Data flow analysis actually begins during basic block optimization, when it is convenient to gather information about the use and definition of values. This information, which takes aliasing into account, is squirreled away in the kill/gen bit matrices. These matrices provide a bit vector for each basic block in the function. Another structure maps bit positions into quadruples which assign values. A set bit in the kill matrix means that in this basic block, the value assigned in the indicated quadruple is overwritten. A set bit in the gen matrix means that the indicated quadruple assigns a value in this basic block which reaches the end of the block.

Data flow analysis combines the definition/use information found above, with control flow information computed earlier, to produce reaching definitions. This information is stored in the in/out bit matrices, and used for loop optimization and register

---

[8]J. Cocke and J.T. Schwartz, Programming Languages and their Compilers: Preliminary Notes, New York, New York University Press, 1970.

---

allocation.

The in/out matrices are identical in organization to the kill/gen matrices above, but the interpretation of their content differs somewhat. A set bit in the in matrix means that the value assigned in the indicated quadruple reaches the beginning of this basic block. A set bit in the out matrix means that the value assigned in the indicated quadruple reaches the end of this basic block. The algorithm which computes the in/out matrices is from Aho and Ullman's text, with little modification.

Before loops can be optimized, they must be found. As mentioned before, the parser does not assist this process. Instead, loops are reconstructed from the flow graph. This method guarantees that all loops will be properly recognized, even those constructed by goto statements.

To detect loops, the global optimizer constructs a depth-first spanning tree (DFST) from the control flow graph and calculates the depth-first ordering. In simple terms, the DFST is a tree representation of the function which starts at the entry point and follows the successor relationships, but without backtracking to basic blocks already visited.

From the DFST, loop entry points (dominators) may be readily found. One basic block is said to dominate another when control flow to the latter in all cases passes first through the former. A loop's entry point dominates all basic blocks in the loop. The loop itself is recognized by a jump from some basic block to its dominator -- this is called a "back edge" and appears in the flow graph, but not the DFST. The global optimizer uses this method to create a list of loops, which is passed to the next phase. The algorithms used here are once again from Aho and Ullman's text, except for first dominators, which are found more efficiently with Lengauer and Tarjan's algorithm[9].

## 6.6. Loop Optimizations

Programs spend most of their time in loops. Therefore the most effective optimizations are accomplished by reducing the amount of code which occurs inside of loops. The global optimizer employs two such techniques: invariant code removal and the elimination of induction variables. The algorithms are taken from Aho and Ullman's text. Loops at all nesting levels are optimized, from the inside out.

Invariant code removal consists of identifying the computations in a loop which do not vary during execution of the loop,

---

[9]Lengauer, T. and Tarjan, R.E., "A Fast Algorithm for Finding Dominators in a Flowgraph", ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, July 1979.

and placing these computations outside of, and before the loop. Here, the result will be computed only once, instead of potentially many times. Reaching definitions computed earlier are used to identify quadruples whose operands are defined outside the loop and are not redefined inside the loop. These are marked and the process repeated in order to account for any ripple effect of the invariant computations.

For each invariant quadruple, checks are made to establish that removal will not affect program correctness. For instance, if the resulting value is live outside of the loop, then its computation may be placed before the loop only if it dominates all loop exits (assuring that it is always executed). Quadruples satisfying these conditions are moved to a new basic block placed just before the loop header.

For example, consider the code in figure 5. The optimization algorithm first finds that T0 and T1 are invariant, and then that T2 is consequently also invariant. These computations are performed in all cases, and occur before all instances of their use. Therefore they can be removed from the loop, and placed before it.

The second loop optimization is done on induction variables. An induction variable is one which may be expressed as a linear function of another variable during the course of loop execution. The global optimizer uses the loop invariant information collected above, and reaching definitions, to identify such

#### C Source Code

```
while (p < &A[top]) *p++ = 0;
```

#### Original Quadruple List

```
L1:    T0 := addr(A)
       T1 := 2 * top
       T2 := T0 + T1
       T3 := p >= T2
       if T3 goto L2
       *p := 0
       T4 := p + 2
       p := T4
       goto L1
L2:
```

#### Optimized Quadruple List

```
L0:    T0 := addr(A)
       T1 := 2 * top
       T2 := T0 + T1
L1:    T3 := p >= T2
       if T3 goto L2
       *p := 0
       T4 := p + 2
       p := T4
       goto L1
L2:
```

Figure 5. A loop-invariant optimization.

---

variables. Typically, the computation of an induction variable can be reduced from a multiplication to the addition of a constant.

### 6.7. Global Register Allocation

Register allocation actually occurs in all phases of compilation. The parser allocates the lower eight CPU registers and the lower four floating point processor registers as needed by functions calls, for parameter passing. The code generator also borrows from the lower half of the register set, for temporary usage. The global optimizer allocates registers from the upper half (excluding the stack pointer) to improve performance. These optimizations resemble source code register allocations, but are more precise. For instance, the global optimizer can assign global registers for temporary variables, which do not exist at source level.

Procedure calling conventions for the System 8000 guarantee that registers R8-R14, and FR4-FR7 will not be changed by a function call. Therefore register allocation is intra-procedural, and the registers from this set which are allocated must be saved on the stack.

The algorithm used for global register allocation is a graph coloring scheme similar to a method described for PL/1[10]. This scheme creates a graph for the function, where each node represents a computed value. Not all values are automatically included; prior optimizations mark likely candidates for register allocation. Using reaching definitions and aliasing information, the algorithm creates an edge between simultaneously live values. Next it tries to color the graph (seven colors for the seven available global registers) so that no connected values have the same color. If coloring fails, then it creates "spill code," which reduces the number of values competing for registers, and repeats the process.

The allocation algorithm decides which values to spill first by choosing those with the lowest weighting factor. A value's weight is calculated beforehand from three components. First a count of references to the value is taken. Where references occur inside loops, they are multiplied by ten times the nesting level. If the value occurs in an explicitly declared register variable, its weight is increased an order of magnitude.

When register allocation is finally decided, the optimizer adds pseudo-operators to the quadruple list which declare register allocation and specify when to load and save values in

---

[10]G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins and P.W. Markstein, "Register Allocation via Coloring", Computer Languages, Vol. 6, pp. 47-57, 1981.

registers.

## 7. Changes to Code Generator

The code generator was modified only as required to support global optimization. It has to understand the modified IL the new parser generates, and also handle the IL extensions added by the global optimizer. As mentioned before, pcc's code generator worked mainly on expression trees, and simply passed through a fair amount of assembly language copy lines generated by the parser. The new code generator has more work to do. It must generate code for the new expressions created for switch statements and functions, and translate the new jump and label line types. The code generator now obtains data definitions from a separate file, and refers to the symbol table for allocation information.

Temporary register allocation in the code generator is essentially unchanged, but global register allocation was extensively reworked. When the global optimizer has not been invoked, the code generator allocates global registers according to explicit declarations in the symbol table. Otherwise the code generator follows the instructions placed in the IL by register allocation in the global optimizer.

## 8. Performance

Figure 6 shows the results of several compute-bound benchmarks when compiled with pcc versus the global optimizing compiler. All were run on a System 8000. The time shown is the user-mode CPU time. The column to the right shows optimized performance as a percentage of pcc performance.

Sieve is a very popular benchmark which re-appeared just

---

| <u>benchmark</u> | <u>pcc</u> | <u>glob</u> | <u>glob/pcc</u> |
|------------------|------------|-------------|-----------------|
| sieve            | 4.74       | 2.36        | 49%             |
| sieve (reg)      | 2.02       | 2.26        | 111%            |
| ack              | 52.0       | 48.1        | 92%             |
| ack (reg)        | 63.2       | 48.5        | 77%             |
| puzzle           | 18.4       | 9.0         | 49%             |
| puzzle (opt)     | 6.9        | 7.1         | 103%            |

Figure 6. Some representative benchmarks.

---

---

recently in Byte magazine[11]. It is commonly improved by register declarations, so results of both versions are shown. The improvement in the version without register declarations is due primarily to global register allocation. When registers are declared, no improvement is possible. Because the global optimizer sometimes creates extra temporaries, performance in this case decreases. Its original performance can be recouped with the global optimizing compiler by invoking only the peephole optimizer.

Ack was mentioned in the introduction to this paper. The benchmarks show a performance improvement which cannot be obtained with changes at the source level. In fact, register declarations make this program substantially slower. This is because global registers must be saved on the stack to make registers available for the declared variables, negating any improvement gained by the register allocation. The globally optimized version works faster in both cases, by minimizing unnecessary moves among register.

Puzzle is another popular benchmark, written by Forest Baskett, and also occurs in two versions. The first is straightforward code, and the second is hand-optimized. Most improvements by optimization in the first version are due to removal of loop-invariant computations in the single function `fit()`, which is listed in the introduction to this paper. As stated in the introduction, the global optimizer achieves about the same results without modifying the source code. The hand-optimized version achieves a somewhat greater improvement which deteriorates slightly after global optimization.

## 9. Conclusions

The global optimizing C compiler's efficacy depends greatly on the nature of the source code it is given. Experience gathered thus far with real-life C programs shows that the most effective optimizations are loop-invariant removal and global register allocation. These optimizations achieve the most dramatic effect on straightforward code which emphasizes structure and clarity. On the other hand, when a program has been hand-optimized to the fullest extent possible with the C language, then the global optimizer makes less significant improvements, or may actually increase execution time. Fortunately, its design allows the programmer to disable global optimization when this occurs, and still take full advantage of the peephole optimizer.

---

[11]Gilbreath, J., and Gilbreath, G., "Eratosthenes Revisited", Byte, January 1983.

Chairperson: *Jim McGinness*  
University of California, San Diego

## UNIX Research at Lucasfilms

*Jim Lawson*

Computer Research and Development  
Lucasfilms Ltd.  
P.O. Box 2009  
San Rafael, CA 94912

Lucasfilms is the creator of such films as Starwars and Star Trek. Mr. Lawson, head of the Computer Research and Development Division, related the current activities of Lucasfilms and the role UNIX plays in them. Deeply involved in the Systems Project whose chore is the maintenance of the many computer systems at Lucasfilms, Mr. Lawson outlined the increasing dependency the company has on computers.

From administrative chores such as word processing, mail, and archiving, to the Industrial Light and Magic Division special effects and film post production, to the Computer Division graphics, game production in tandem with Atari, the computer and UNIX are integral to Lucasfilms.

Currently utilizing four 11/750's, one 11/780, and a network of 68000's, the company is awaiting VAX 4.2BSD support. John Seamons and Bill Reeves have created an extended file system with network-wide access to all files utilizing ethernet which requires no local discs.

Lucasfilms is active in digitizing as much of the process of movie production as possible; from their Audio Project which uses a 68000 as a controller for their audio signal processor, to their Video Project whose main goal is discovering better ways of utilizing video technology in a film oriented domain.

Lucasfilms has found UNIX an ideal operating system as it has allowed a uniform environment for the many different activities presently occurring in the company.

## ARIEL: An Experimental UNIX-based Interactive Video Information System

*R. C. Haight and D. B. Knudsen*

Bell Laboratories

ARIEL is an experimental UNIX-based interactive video information system designed by Bell Labs for Disney World's EPCOT Center. The system consists of laser video discs, video overlay graphics, a tune synthesizer, touch screen input devices, phone connections, and up to fifty user terminals located throughout the Park.

Users interact with the system by touching the menu displayed on the touch sensitive screens; information which can be accessed ranges from current events to geographic history. Software for the system includes routines which interface to the above mentioned hardware via RS232 connections, shell level utility commands



---

which use these routines to perform tasks on the devices, a music compiler written in C, a *yacc*-based script compiler used to compile the language which controls the devices, a script interpreter consisting of compiler output plus additional files, and high level control programs to handle interprocess and intermachine communications. The script language used is based on a single data type (strings) and includes many device primitives — video, audio, touch — as well as a C pre-processor used to define macros.

UNIX as a base allowed the development of the system in a timely manner with relatively few people.

---

# **Development of a Digital Simulation System in a UNIX Environment**

*William Raves and James Cassidy*

Computer Automation  
2102 North Forbes Boulevard  
Suite 107  
Tucson, AZ 85745

Computer Automation has been able to utilize UNIX in the development and implementation of an applications system used in the design and testing of printed circuit boards. The system consists of a slave processor "Digital Simulation System" which runs UNIX, linked to a resource manager hooked up to numerous single user cpu's dedicated to testing. Simulation and testing involves defining the elements and topology of the particular circuit board, performed at the simulating stations; once simulated, the model is tested for possible faults at the testing stations.

In that much of the process of design and testing is based on feedback, the goal of the software engineer was ease of modification. UNIX lends itself well to this; by using shell script for control flow Computer Automation arrived at the flexibility that was necessary; by converting these shell scripts to C programs, increased processing speed was attained.

Many of the concerns of Production Management were quelled by the UNIX Operating System. The hierarchical file system aided in the complexities involved in a multi-shift production environment allowing the loading of appropriate file systems for each shift. In addition, the shell environment proved useful to the security needs of the management.

UNIX allows a flexible development environment and control of that environment; in addition, the portability of UNIX allowed the development of the software on a totally different machine, cutting a significant amount of set-up time.

---

DEVELOPMENT OF A DIGITAL SIMULATION SYSTEM  
IN A UNIX ENVIRONMENT

William Raves  
James Cassidy  
Computer Automation, Inc.  
Industrial Products Division  
2102 N. Forbes Blvd. Suite 107  
Tucson, AZ 85745 (602) 622-2513

ABSTRACT

This paper discusses the exploitation of features provided by the Unix\* operating system in the development and implementation of an applications system designed to perform logic simulation of printed circuit board (pcb) descriptions and to monitor the results of pcb testing.

Special emphasis is given to the utilization of the Unix file system in controlling and simplifying the applications environment, to the use of the Unix shell in the development of a flexible user interface, and to the utility of multi-tasking and spooling in the scheduling of activities in a production environment.

---

\* Unix is a trademark of Bell Laboratories

\*\* Marathon is a trademark of Computer Automation, Inc.

---

## INTRODUCTION

---

During the course of the past few years, we have witnessed a phenomenal growth in the acceptance and implementation of the Unix operating system and Unix clones in the computing world. Since this turn of events may be said to speak for itself, there seems little need to reiterate the widely-publicized features of Unix which make it so attractive. Much of this attention, however, focuses on the powers of Unix in the service of software development systems and much less attention is paid to Unix as an environment for specific commercial applications. There may be considerable advantages in developing a particular application in one environment and distributing that product in the same environment. As Unix continues to spread through mainframes and mini-computers, more critical analysis should be brought to bear upon those experiences of applying Unix to a variety of computing tasks not strictly related to software development.

This paper discusses the exploitation of features provided by the Unix operating system in the development and implementation of an applications system designed to perform logic simulation of printed circuit board (pcb) descriptions and to monitor the results of pcb testing. This project employs Unix both as a development system environment and as an application environment. Our emphasis will be primarily on features of the simulation system and secondarily on general software development topics. Special attention will be paid to the contributions of Unix toward:

- (1) the development of a flexible user interface
- (2) control of the application environment
- (3) management of a remote processor

We will conclude with some general comments and impressions regarding the use of Unix in our development efforts. Since the simulation system is relatively new, this analysis cannot, unfortunately, benefit from the added dimension of extensive user response and experience.

## OPERATING ENVIRONMENT

---

The digital simulation system is one part of a comprehensive multi-user pcb test system under development by the Industrial Products Division of Computer Automation. The system, known as "Marathon" \*\*, is directed toward the high end of the automated test equipment marketplace. One of Computer Automation's multi-user processors, referred to as the "resource manager", is the kernel of the Marathon system. Several remote processors may be tied directly or indirectly into the network.

Stand-alone processors which perform the actual testing of printed circuit boards are connected to the resource manager by a high-speed link over which data files prepared after simulation are sent to the testers and the results of testing are communicated to a database maintained by the resource manager. Additional processors which are dedicated to digital simulation are entirely

§

---

slaved to the resource manager. Only the simulation processors will be discussed here, since they are controlled by the spooling mechanism provided through Unix.

The process of simulating the digital components of a printed circuit board begins with the creation of a circuit description detailing the components of the board and their interconnections. From this description, a mathematical model of the circuit is constructed into which all possible shorts and open faults are inserted. During actual simulation, a set of user-supplied stimulus patterns specifying the digital signals to be applied to the circuit model is used to exercise the logic of the board in such a way that a high percentage of the inserted faults will be detected. During refinement of the stimulus pattern set, a great deal of editing, simulating, and interactive analysis takes place.

Digital simulation involves a somewhat balanced mix of operator-intensive and machine-intensive activities. The development and implementation of such a system demands an environment that offers an efficient user interface as well as efficient machine management.

#### DEVELOPMENT OF A USER INTERFACE

---

The development of a human-engineered user interface is a continuing process that benefits most from experience and feedback. For this reason, one of the primary design goals of the interface was to generate a flexible system that could be modified easily and quickly. The format of the interface reserves the bottom four lines of the screen for program titles, status messages, and blocks of software-coded function keys and their labels which correspond to designated keys on the keyboard. The remainder of the screen is dedicated to program output. The result is a key-stroke-driven interface, applied to virtually all software modules in the simulation system, that is consistent, easy to use, and easy to comprehend.

Unix command language "shell scripts", in conjunction with simple in-house utilities for controlling the crt screen, were employed heavily during the initial phases of the implementation. The use of shell scripts provided two distinct advantages toward realization of the design goal. The first was the relatively short development time required to implement modifications, primarily because those pieces of software most likely to undergo revisions were coded in a language which did not have to be compiled. The result of this situation was that design and development could proceed in parallel and experimentation was encouraged simply because change was cheap. The second advantage was the low maintenance costs of the entire interface system, attributable to that fact that very small utilities were used and that the Unix shell command language itself is a simple but surprisingly powerful high level language. Personnel could enter the project and bring themselves to productive levels in a short period of time.

It should be emphasized that the use of the shell language in the user interface project was confined to the development stage since the performance of the early system was inadequate for an interactive system in the real world. The task of converting from the shell-based system to a C-based system was, however, a

---

simple and rapid process. The screen control utilities, whose reliability had already been confirmed, were transformed to C functions with minimal change to the code itself. Conversion of the interface driver was equally efficient since the control flow and basic configuration of the system were fairly stable by this time.

#### CONTROL OF THE APPLICATION ENVIRONMENT

---

The friendliness of the user interface was an important consideration in the design of the simulation system but this aspect of the system could not, by itself, guarantee success. The process of actually simulating a pcb had to be as simple as possible and at the same time, conceal as many of the details of the implementation as possible. These two goals, simplicity and abstraction, were approached with the assistance of several operating system features provided by Unix.

The first of these is the hierarchical file system. The organization of data files, made possible through the existence of directories, offered a ready-made framework which could be tailored to match the logical organization of data for a particular circuit board. The result is a consistent organization with consistent file names that applies to all circuit boards that may be maintained on the system. This uniformity facilitates maintenance and trouble-shooting and renders global updates to pcb data feasible. In addition, the highest levels of the application hierarchy, where circuit boards and component device libraries are defined, could be configured as independent file systems. Major production shifts in the testing environment could be reduced to unmounting and mounting the appropriate file system. This type of global change can be signaled to the system simply by altering the values assigned to exported shell variables which indicate the file system roots of various aspects of the application environment.

The file protection mechanism of Unix did not play a great role in the development or implementation of the simulation aspect of the system, apart from insulating system software and control data files from inadvertent abuse. But its role is magnified when the simulation system is viewed as one segment of the total system. The participants of the production environment with access to the system range from data entry personnel to management personnel and some installations will undoubtedly demand restrictions on the availability of data and function based upon the requirements of a particular job. The uniformity in access privileges evident in the simulation system belies its importance to the production environment as a whole.

The association of a time-of-last-modification with each file in the file system greatly enhanced the opportunities for automating the process of simulation. From the system's point of view, simulation is basically a linear process on a system-maintained body of data. At any point in this process, the next state of the data depends upon a combination of the current state, user inputs, and the next operation itself. From the user's perspective, the process is generally linear but punctuated with significant loops in which the user is varying some set of inputs which he can supply to the system. Such loops occur when the user is constructing and verifying a circuit database or when the pattern set driving

---

simulation is repeatedly refined in order to attain an acceptable level of fault detection. Since any one of several input sets may be altered by the user in any one of these loops, the system software can use the last-modified dates provided by Unix in determining whether or not it is necessary to execute some data transformation. The result is a system which performs only when necessary and can do so without user intervention.

File system dates are also proving valuable in tracking down problems with the system. They provide a clear indication of the progression of a process which may create several files throughout its execution.

Several other miscellaneous features of Unix proved useful in implementing a simulation system and controlling access to such a system. The shell execution "path" variable limits accessibility of system software to those users who require it and at the same time reduces the time necessary to search for and validate a request for command execution. The shell "home" variable initially places each user at an appropriate node in the file system tree. This location serves to define a subtree over which the user has more or less exclusive control and, by default, the remainder of the file system over which the user may or may not have some degree of control. Finally, Unix system configuration files such as "etc/passwd" offer the facility of defining system access based upon the user's entry point into the system and the program that will be executed immediately after the login protocol is successfully completed. Using this capability, test engineers using the simulation system would find that their initial contact with the system is through the function key interface driver implementing the simulation functions, while managers or clerical personnel might have a totally different functional capability after logging in.

In short, the simulation system demonstrates that ordinary users need not have any contact at all with the operating system. This is a distinct advantage with respect to the initial productivity of new customers and of new personnel who may be hired later on, providing that a comprehensible and simple interface is available. This can be an even greater advantage in a system not specifically designed for commercial applications which is unforgiving and often reticent. The features of the Unix operating system mentioned above proved to be powerful tools in the quest for abstraction. They aided not only in concealing the details of Unix but in concealing many of the details of simulation itself.

#### MANAGEMENT OF A REMOTE PROCESSOR

---

Production throughput during the simulation phase of pcb testing is enhanced by confining interactive activities to the multiuser processor running Unix and by confining actual simulation, which is a machine-intensive non-interactive computing task, to a single-user cpu running a local operating system. The benefits of this arrangement are made possible by a number of Unix features including the ability to initiate asynchronous processes, the uniformity of file and device i/o, a descendant process structure which allows a predecessor to suspend until completion of child processes, and use of the mail facility to achieve process-to-user communication. In the implementation described here, the user initiates a simulation by specifying a board name, a file of stimulus patterns, and any other relevant parameters. Data transfers, job selection, and actual

---

execution are then handled by the system in the background and the user is informed via mail when the simulation is completed.

The easiest method of interfacing a remote processor in a Unix environment is via a standard serial tty link which is the supporting link for most Unix-to-Unix communications. Since the remote processor, in this case, is not a Unix machine and does not support a C compiler, a reliable communications protocol for data transfer was developed and implemented. A standard ACK/NAK protocol with LRC checking is employed to control and verify pre- and post-simulation data communications. A high-speed link with mega-baud throughput is currently being installed. The tty link, however, still provides a management interface to the simulation processor and proved to be an indispensable communication medium since it permitted verification of system integrity during the early phases of development.

Because the remote processor is entirely slaved to the resource manager, it may be necessary to modify its local operating system to some extent. The areas of concern are concentrated in the terminal driver resident on the slave, since the host controls data transfers, and in the operator command prompt sequence since the controlling process on the host is also a command driver for the remote machine. In the configuration described here, very few modifications were required due to the fact that the slave operating system employed a command prompt sequence that was unique for all software running on this machine. Thus, synchronization of activity on the simulation cpu could be accomplished by using its standard operator prompt sequence. Only slight modifications to the tty driver were necessary involving the filtering of cursor control characters which had a negative impact on data transfers.

The management software described above is enclosed in an application layer which provides for the spooling of remote tasks in a batch-like mode. The spooler software is quite similar to existing spooler mechanisms in Unix such as that controlling a hard-copy printer (lpr), but it also includes a number of extensions such as job priorities, target machine designations, and user access/control over initiated jobs.

#### COMMENTS

The recent proliferation of the Unix operating system attests to the power of Unix in the service of computing tasks simply because it provides an environment in which the performance of those tasks is facilitated. We found Unix to be an excellent development environment for many of the same reasons mentioned elsewhere: efficient execution, a file system saturated with abstraction capabilities, an array of easily-coupled software tools, etc. It is apparent when working in such an environment that it was designed with software development in mind. But the purpose of the discussion above was to provide some indication of the effectiveness of Unix in an application oriented environment.

Toward this end, Unix proved to offer a more than adequate measure of flexibility with respect to the organization and configuration of the simulation system. The file system was particularly helpful in organizing data relevant to the simulation of circuit boards and in aiding the automation of pre- and post-



---

simulation data transformation processes. The concept of mounted file systems and the ability to bind path prefixes to shell variables reduced major shifts in the production emphasis to nearly trivial operations. Background execution helped relieve the dilemma posed by a system which combines highly interactive and time-consuming, non-interactive activities. The uniform treatment of and direct control over peripheral devices tied to the resource manager permitted the dedication of one or more cpu's to simulation itself. Finally, the Unix mail facility provided a means of communicating process completion and status for those activities controlled by a spooler or otherwise executed in the background.

We would be naive to admit to no difficulties in our efforts with Unix and it may be worthwhile, at this point, to mention several of the more troublesome.

The simulation system described above was a pre-existing system operating on a single-user cpu. Much of the data design of this earlier system was constrained to economize disk and memory usage. The new system, for reasons of development costs, was a mixture of wholly redesigned software and software that was untouched from the earlier version. Interfacing the new software and data designs with the old proved difficult in the early development stages since much of the data to be processed was non-ascii and Unix provides no utilities for processing such files. Instead, all of the text processing tools available expect a line-oriented structure. It proved easy enough to compensate for these shortfalls by implementing in-house utilities, but the development momentum suffered somewhat in the effort.

It was mentioned previously that the control which Unix can provide over the file system, and hence the information contained in the system, was a distinct advantage. But we might ask ourselves whether it is Unix which creates the need for control. The sociable use of the Unix operating system in a situation where many users require access to common data (such as pcb data) demands some knowledge and care if costly setbacks are to be avoided in a production environment. In lieu of education, the application software must intercede to ensure protection. In a purely software development-oriented environment such a requirement is less important since it is incumbent upon the users of such a system to fully understand it and employ it in an atmosphere lacking all but self-imposed restraints. The application domain is another matter, since in many cases it is unrealistic and unnecessary to expect the users to comprehend the full measure of software development potential lurking in the system. In addition, we can expect a more heterogeneous user population in terms of skill level, functional concern, and attitude toward the system.

Nevertheless, the resulting system seems to offer the best of both worlds. On the one hand, the simulation system represents a controlled environment tailored to a specific test installation where the application software serves to hide the operating system details from the user and restrict the user's access to other aspects of the system. On the other hand, the full power of Unix is available to those who have the need or the willingness to exploit it.

---

## Meeting the Coming UNIX Training Challenge

*Jay R. Hosler*

User Training Corporation  
P.O. Box 970  
Soquel, CA 95073

Like many, User Training Corporation envisions an explosive future for UNIX; the already apparent influence this operating system has had on the industry could easily spread to retail sales systems, office automation, and even home computing, making UNIX the operating system choice among the 16 bit microprocessor systems.

A prerequisite for this growth is a cost effective means of training the large number of new users. The past has provided training via seminars, personal instruction, and commercial books. User Training Corporation has come up with a novel training technique utilizing the medium of Audiodigital Recordings.

Audiodigital Recordings pair up an audio track with a track of digital information to provide the illusion of someone using the system while explaining what he is doing. They call this "tutor simulator." Much more cost effective than a personal tutor, the Audiodigital method does not require a computer; once encoded, all that is necessary is a digital converter and a terminal.

Audiodigital recordings were devised at Stanford over five years ago but User Training Corporation is the first to exploit this innovative medium for demonstrating the use of interactive systems. User Training Corporation has a continuously growing library of UNIX tutorials and recognizes the potential for this medium in all sorts of training needs.

---

## MEETING THE UNIX\* TRAINING CHALLENGE

Jay R. Hosler  
User Training Corporation  
P. O. Box 970  
Soquel CA 95073

### I. INTRODUCTION

Suppose you had to learn UNIX for the first time. How would you choose to be trained? By the Bell manuals? By classes? By commercial books? Probably none of the above; you would probably choose a personal tutor. Failing that, you would probably find an expert user and watch him work, asking questions as necessary. Nothing teaches how to interact with a computer system as well as example.

This talk is about training people to use UNIX. It is not theoretical. I'm a practitioner -- a developer of training materials. The talk focuses on the novel and powerful training medium I use: audiodigital\*\* recording.

Audiodigital training presentations teach not so much by conventional explanation as by example. Viewing one is much like learning from a personal tutor. I believe that audiodigital recording has the potential to become the training medium of our time.

### II. THE UNIX TRAINING PROBLEM

Only a few years ago, UNIX could be found only inside Bell and in universities. Now suddenly it has entered a new phase of its existence and is the subject of much attention. UNIX could become the dominant operating system for 16-bit microprocessors and could become popular even in retail stores. There are many implications: UNIX will do more kinds of jobs; it will be used by many more people; and it will be used by an entirely new kind of user -- the computer novice.

One of the effects of all of this is a sharp change in the kind of training that UNIX users need. Traditionally, UNIX users have trained themselves informally using the buddy system: find a buddy and watch him use the system. The manuals are used for supplemental reference. The formally trained user -- a rarity -- learned at a seminar. These techniques won't answer the new need. Seminars are expensive and inconvenient, and their effectiveness isn't uniform. The buddy system is labor-intensive and disorganized. And the Bell manuals, while comprehensive, are hardly tutorial.

The new UNIX user community needs a training technique for training great numbers of new users cost-effectively. An even

---

more critical new requirement is the training of computer novices. UNIX is used increasingly by secretarial and clerical personnel, and retail customers are primarily novices. Novices are so unfamiliar with UNIX that they can't absorb it easily while watching buddies work. To overcome their keyboard fear, novices need some personal handholding.

The requirement for a non-labor-intensive training method that offers enough personal tutorial help to novices to get them started suggests some kind of "canned" training. But canned training products have earned a reputation for low quality. Among other things, they often put their users to sleep, a condition in which learning is markedly retarded.

But watching a system in use, if you have a grasp of its fundamentals, is not boring. Watching a user do his work is an experience completely different from listening to a lecturer explain how to use a system. Thus audiodigital recording, with its emphasis on example, is well suited to the new UNIX training need.

### III. TRAINING AND THE AUDIODIGITAL MEDIUM

Audiodigital recording is a hybrid audiovisual medium in which the visuals are displayed on the terminal rather than on a TV monitor. An audio cassette tape carries both the audio and digital information. In the basic form of audiodigital recording, both channels of the tape are recorded simultaneously: a microphone captures the words of the instructor, along with background noise such as keystroke sounds. While this audio information is recorded on one channel of the tape, the digital traffic between the terminal and the system is recorded on the other. The resulting tape contains a live recording of an online session.

Audiodigital playback is unique in that it gives a strong illusion of someone interacting with a system, explaining what he's doing. The medium is thus excellent for capturing teaching performances. It is at its best doing annotated demonstrations of real sessions -- just where many media are weakest. The novelty of the presentation enhances the training process by reducing the danger of boring the audience.

The medium is highly personal. Typically, a single trainee views a recording, using a headset and his own terminal. The resulting intimacy contributes significantly to the training effect, as does the similarity between the training environment and the real working environment.

---

## History of Audiodigital Recording

Audiodigital recording technology is not truly new. Several universities and companies have experimented briefly with similar techniques. They presumably thought initially that the simple technology meant simplicity of audiodigital production; they were then probably discouraged when they realized the enormity of the task of developing good presentations. Additionally, there are many technical issues that greatly complicate the production of audiodigital recordings on a large scale. It thus seems likely that the early experimenters dropped their audiodigital projects when they realized that good presentations were beyond their resources.

At Stanford, the medium had a more productive history: it was pursued to a doctorate by Jan Bleil, starting in 1976. She began by making six hours of courses for Stanford's WYLBUR system, which is a little like beginning the study of a new language by writing a novel in it. Stanford has used the WYLBUR tutorials ever since to train literally thousands of people. Jan Bleil went on to a detailed examination of audiodigital authorship -- the problem area that the other early workers had ignored. By 1980, she had developed a process for producing high quality tutorials.

The other early projects have yielded a few products on the market now that are technologically related to audiodigital recording. But none uses the medium to record interactive sessions or to teach people to use computers. It is fair to say that the development of audiodigital recording into a viable computer training medium began at Stanford. It continues at User Training Corporation, of which Dr. Bleil is a co-founder.

## Audiodigital Recording as a Training Medium

Several factors contribute to the training effectiveness of audiodigital recordings. First: they teach by example, which for interactive systems is far superior to explanation alone. Viewing an audiodigital recording, a trainee sees his own terminal behave just as it will when he uses it to interact with the system. And that verisimilitude -- the extraordinary likeness between the training environment and the work environment -- is unique to audiodigital recording.

Second: audiodigital recordings provide spoken instruction. The audio component is critical to the learning process in the majority of the population. While audio is not unique to audiodigital recording, the spoken word contributes significantly to the power of any training technique.

Third: audiodigital recordings are inherently self-paced. The trainee can interrupt the course anytime to repeat a section or take a break. Returning, he can restart the course anywhere.

---

Also, the course can begin at the trainee's convenience. Of course, most canned training could be self-paced. Unfortunately, many courses are intended for group use, which defeats self pacing. We design our courses for individual viewers because the pacing issue is so important to effective training. Fortunately, the medium lends itself well to intimate, one-to-one tutorials.

Viewing an audiodigital recording is a non-interactive activity. There is much confusion about the role of interaction in training. For most people, the training process has two important components: one in which they absorb new material, and one in which they reinforce it by practice. In our courses, trainees practice by doing carefully constructed exercises. While viewing the audiodigital tapes, they aren't practicing -- they are absorbing new material. Frequent practice sessions are designed into the courseware. In other words, viewing an audiodigital tape is a different kind of experience from practicing -- and an equally important part of the training experience.

Moreover, the many training courses that consist solely of interactive software, besides having no audio, can confuse novice users. Novices find it conceptually difficult to separate a new system into its intelligent components. (A UNIX example: novices are often confused by the important notion that it is the shell that expands wildcard filenames.) The presence of interactive teaching software can exacerbate this confusion by giving a distorted view of the system being taught. The UNIX 'learn' program is a case in point: it varies the perceived file environment by manipulating the current directory. Further, the learn user sees a 'shell' significantly unlike any other shell. For novices, interactive courseware is best used in combination with demonstrations of the system used exactly as the novice must use it.

There is also an important hidden issue in which audiodigital recordings are superior for training to presentations on TV monitors. We are all utterly accustomed to technical perfection on television. Audiovisual presentations with lower technical values appear amateurish; this impedes the training they do in a subconscious and significant way. Audiodigital recordings, because of their novelty, are not subject to this phenomenon.

Realization of the potential benefits of a new medium is not automatic. In the final analysis, the effectiveness of a tutorial is determined by its overall quality. Development of good audiodigital tutorials requires exacting attention to quality and balance. Technological novelty cannot save a poorly organized script; tutorials on tape require extraordinary attention to pace and development of theme. But this medium provides an extraordinary set of tools to the developer of tutorials. It could become a major factor in the way people are trained to interact with computers.

---

#### IV. THE COURSEWARE DEVELOPMENT PROCESS

The importance of overall quality in training presentations suggests a development process that provides good quality controls. This section briefly outlines how courses are developed by User Training Corporation. The expense of this rather formalized process -- about 250 manhours per hour of finished product -- is justified by the consistently high quality of the tutorials it produces.

A tutorial has four logical components, which become part both of the recordings and of the accompanying workbook:

1. The SCRIPT: a word-for-word transcript of the spoken narrative;
2. The EXAMPLES: the interactive examples that are used during the teaching session, and the files they use;
3. The EXERCISES: the written or programmed exercises that provide practice for the trainee;
4. The BLACKBOARDS: the non-interactive displays that are presented on the terminal during the narrative. Blackboards convey agendas, summaries, incidental annotations, and other ideas.

The process of preparing a course consists in substance of the development of each of these components. User Training Corporation's courseware construction process is modelled after the process of software development. The production of the components of the course is addressed much as a software development project addresses the production of logically related software modules.

The process has four major phases, with a major review after each:

1. Preliminary design: the course architects specify the major layout of the course and fix the list of topics covered in each time segment. They may choose to specify any part of the course to any level of detail; as with software, however, the preliminary design specification normally outlines most of the course only in rough detail.
2. Detailed design: each of the four components of the tutorial is defined in detail. The script is represented by a detailed outline; the examples, exercises, and blackboards are specified in rough but comprehensive form. The workbook layout is specified. This phase corresponds closely to the detailed design phase of a software project.

3. Component preparation: in this phase, which corresponds closely to the coding phase of a software development activity, each of the four components of the tutorial is developed completely. The tutorial workbook is prepared and reviewed.

The "voice" of the tutorial consults with the writers throughout this phase, and periodically test-records the entire tutorial for group review.

4. Recording: The tutorial is committed to tape in small "takes". The final master tape is prepared in a last editing process which is supervised closely by the writers.

#### V. USER TRAINING CORPORATION AND AUDIODIGITAL RECORDING

User Training Corporation was formed in 1981 to make audiodigital training courses for UNIX and other interactive systems. Our first product, called the UNIX SYSTEM TUTORIALS, is entering its second year in the marketplace. These tutorials show that real solutions exist to the major technical problems inherent to the medium. The UNIX product currently includes about twenty hours of courseware -- some designed for novice users and some for experienced technical personnel. The product line will be continuously expanded and upgraded to meet the needs of the market and to keep abreast of UNIX developments.

User Training Corporation is also developing other audiodigital training products, to be offered in retail stores during 1983 for a variety of popular retail software packages, including UNIX.

#### VI. DEVELOPMENTAL DIRECTIONS

1983 will see further audiodigital training developments, along several lines. First: online exercise software will be added to the UNIX tutorials. Currently, the exercises are printed in the workbook. The new interactive exercises will be integrated with the tutorials by using the same file environments in the exercises and the recorded examples. A thoughtful combination of audiodigital recordings with interactive practice software will provide substantially enhanced training.

Second: technological improvements are being made in the medium itself. The UNIX tutorial recordings include all the information that is sent by the system to drive a user terminal. This technique is not applicable to microprocessor systems with memory-mapped CRT displays, because the memory map is a "private" interface between the application program and the display. User Training Corporation is now developing tutorials for the IBM Personal Computer, using a more complex technique for making and playing back audiodigital recordings. The effect is like that of the UNIX tutorials: viewing them is like watching an instructor



---

use the computer.

We are also examining the potential of random-access digitized voice. In theory, random access audio would permit the construction of interactive tutorials with full audiodigital voice response. The courses could be carried either on a standard storage medium or on video disk, which could also support an advanced graphics capability. However, the development of such courseware is very complex. It is likely that commercial production of fully interactive tutorials must await a fuller understanding of the technology of courseware itself. New hardware technology will not significantly hurry that understanding.

The medium is not limited to computer training. Audiodigital recordings make effective sales presentations and proposals; for example, they allow demonstrations of systems that don't yet exist. There are also numerous potential applications within the educational system, related to the current demand for widespread computer literacy.

User Training Corporation is pleased to bring this superb medium to the commercial marketplace. We expect that others will begin to use audiodigital recording, for internal training at first, and then for training products. We want to assist this positive development. The medium can upgrade the training standard throughout the industry. We have considerable investment in audiodigital recordings; we welcome dialogues about broadening their application.

\* UNIX is a trademark of Bell Laboratories

\*\* audiodigital is a trademark of User Training Corporation

Chairperson: *Andy Tannenbaum*  
Bell Labs

## UNIX for the STD Bus

*Luigi Cerofolini*

Istituto Di Matematica Applicata  
Facolta Di Ingegneria  
Universita Di Bologna  
Via Vallescura 2  
40136 Bologna Italy

The STD bus, jointly developed by Mostek and Pro-Log around mid-1978 and now being in the public domain, has gained wide acceptance among designers and, with over 70 manufacturers producing board based products for it, it is one of the fastest growing buses of the past ten years. So it was natural to put UNIX, one of the most popular operating systems available today, on the STD. The CPU choice was very natural. The STD data bus is 8-bit wide while UNIX was originally designed for a 16-bit CPU: The Intel 8088 CPU and associated co-processors 8089 for I/O and 8087 for number crunching seemed to satisfy nicely the general system requirements. Because of the small size of the STD boards (6.5 in. × 4.5 in.) we have been forced to work on a very modular multi-processor system and this turned out to be the key factor in order to have a high performance system.

We divided the system into four main subsystems or Functional Units (FU): CPU, terminals, disk and tape, and memory. In order to offload the CPU from low level peripherals control activities we needed intelligent I/O controllers, but this was contrasting with the very small board size. So we opted for a two-board solution for every FU (except memory—that fits nicely into one STD board): one of the two boards, the Universal Processor Unit or UPU, is the same for all FUs, while the other board, the Special Unit or SU, is tailored to specific functions. The two boards of the same FU are connected together through a flat cable.

This possibility of sharing the same hardware between different FUs was a decisive fact for the success of the project: we had the same well known and debugged piece of hardware (the UPU) as the starting point for all new specialized SUs we needed for the system.

All the UPUs (Universal Processor Units) share the same hardware design and their main components are: the 8088 CPU, one free 40-pin socket (to be used for a coprocessor like the 8089 or the 8087 or for a Silicon Real Time Operating System Kernel like the 80130) whose use is application dependent, latches and STD buffers, logic for DMAing into the system STD bus, local ROM and RAM and a jumper configuration array.

The Special Unit structure can be usually considered as an I/O adapter: for the Terminals Controller it is simply an array of programmable UARTs; for the Disc and Tape Controller it is an adapter to a popular intelligent formatter/controller; for the CPU board it is a Memory Management Unit plus Timer and Interrupt controller logic.

We are now developing controllers for a Laser Printer and for a Graphic CRT and we found the two boards-set approach very powerful and flexible.

All FUs are the same to the operating system and this uniformity has various nice consequences like easy maintenance, short design time, good performance and low cost.

---

This page intentionally left blank

---

## **Ctrace - A Portable Debugger for C Programs**

*J. L. Steffen*

Bell Laboratories, Room 2C-331  
Naperville-Wheaton Road  
Naperville, IL 60566

Ctrace is an easy to use, portable C language debugging tool that lets you follow the execution of a C program, statement by statement. The effect is similar to executing a UNIX shell script with the `-x` option. Ctrace is far easier to use than *adb* or *sdb* and, unlike these debuggers, it is completely machine independent so it is highly portable. It has even been used to debug C programs in non-UNIX environments.

---

## Ctrace - A Portable Debugger for C Programs

J. L. Steffen

Bell Laboratories  
Naperville, Illinois 60566

### ABSTRACT

Ctrace is an easy to use, portable C language debugging tool that lets you follow the execution of a C program, statement by statement. The effect is similar to executing a UNIX™ shell script with the `-x` option. Ctrace is far easier to use than Adb or Sdb, and unlike these debuggers, it is completely machine independent so it is highly portable. It has even been used to debug C programs in non-UNIX environments.

### 1. INTRODUCTION

The UNIX time-sharing system, which provides excellent facilities for most aspects of software development, surprisingly lacks an easy to learn and simple to use debugger. As a result, most programmers debug a program using the primitive method of inserting print statements into the program. This is time consuming because it takes many statements scattered throughout the program to get enough information to isolate a bug. The print statements themselves can cause syntax and other errors, and are time consuming to add and delete.

The debugging tools provided by UNIX, Adb and Sdb, are so machine dependent that it takes a major effort to rewrite them for a new computer. Adb is primarily suitable for examining the contents of variables and the stack in a core dump after a program aborts. It is of little value for debugging C programs because breakpoints can only be inserted at the machine code level. Sdb is an enhanced version of Adb that is not available for the PDP-11 computer, thus severely limiting its usefulness. It allows the insertion of breakpoints at the C statement level, thus allowing the equivalent of print statements to be inserted without recompiling. However, the programmer must learn a whole new language for controlling the execution of the program.

Ctrace, however, is an easy to use debugging tool for C programs that displays all the information necessary to find a program bug, without the need for interactive commands and without overloading the user with information.

### 2. DESCRIPTION

Ctrace is a preprocessor that inserts source language debugging code into the program before compilation that will print the debugging information at run time. This makes Ctrace machine independent, unlike the UNIX debugging tools, so it can be used to debug C programs on any machine for which a compiler or cross-compiler exists. Furthermore, the machine does not have to be running the UNIX operating system.

---

As each statement in the program executes it will be listed at the terminal, followed by the name and value of any variables referenced or modified in the statement, followed by any output from the statement. Loops in the trace output are detected and tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed every 1000 times through the loop to help you detect infinite loops.

You can trace the entire program or just certain functions. Within a traced function, you can turn the tracing on or off by putting `ctron()` and `ctroff()` function calls in the source code. This lets you skip the initialization of an array, for example.

If you are familiar with a breakpoint debugger like Sdb, you can also turn the tracing off or on by inserting breakpoints that set the `tr_ct` variable to 0 or 1, respectively. This gives dynamic control of the tracing.

### 3. EXAMPLE

If the file `lc.c` contains this C program:

```
1 #include <stdio.h>
2 main()          /* count lines in input */
3 {
4     int c, nl;
5
6     nl = 0;
7     while ((c = getchar()) != EOF)
8         if (c == '\n')
9             ++nl;
10    printf("%d\n", nl);
11 }
```

and you enter these commands and test data:

```
cc lc.c
a.out
1
(CTRL-D)
```

the program will be compiled and executed. The output of the program will be the number 2, which is not correct because there is only one line in the test data. The error in this program is common, but subtle. If you invoke Ctrace with these commands:

```
ctrace lc.c >temp.c
cc temp.c
a.out
```

the output will be:

```

2 main()
6     nl = 0;
    /* nl == 0 */
7     while ((c = getchar()) != EOF)

```

The program is now waiting for input. If you enter the same test data as before, the output will be:

```

    /* c == 49 or '1' */
8     if (c = '\n')
    /* c == 10 or '\n' */
9         ++nl;
    /* nl == 1 */
7     while ((c = getchar()) != EOF)
    /* c == 10 or '\n' */
8     if (c = '\n')
    /* c == 10 or '\n' */
9         ++nl;
    /* nl == 2 */
7     while ((c = getchar()) != EOF)

```

If you now enter an end of file character (CTRL-D) the final output will be:

```

    /* c == -1 */
10    printf("%d\n", nl);
    /* nl == 2 */2

    /* return */

```

Note that the program output printed at the end of the trace line for the `nl` variable. Also note the `return` comment added by Ctrace at the end of the trace output. This shows the implicit return at the terminating brace in the function.

The trace output shows that variable `c` is assigned the value `'1'` in line 7, but in line 8 it has the value `'\n'`. Once your attention is drawn to this `if` statement, you will probably realize that you used the assignment operator (`=`) in place of the equal operator (`==`). You can easily miss this error during code reading, and the Lint command, which could warn you that you used an assignment expression in place of a relational expression, unfortunately does not detect it either.

#### 4. EXPERIENCE

In general, users found the trace output to be so easy to follow that even a large amount of output was not overwhelming. However, eliminating the trace output from low-level functions aided bottom-up testing and was a convenience on low-speed terminals. People who spend most of their time fixing problems in existing programs really liked the option to trace only a few functions.

Besides the obvious benefits of debugging, Ctrace has helped people learn how a poorly documented or complicated program works. The

---

purpose of variables with cryptic names often becomes clear when you see them used in an executing program.

The popularity of Ctrace is best shown by its private installation and use on over 190 computers at 17 Bell System locations. Several people first tried Ctrace after being unable to find a particular program bug with Sdb, and their success using Ctrace shows that it is a better debugger in many respects. New C programmers have found Ctrace easier to use than Sdb because you do not have to learn a new command language to debug your program.

Ctrace has proven its portability by being installed on all versions of UNIX used within Bell Labs on PDP-11/70, VAX-11/780, IBM 370, Univac 1100, and Western Electric 3B20S computers. This includes UNIX/RT (formerly MERT) and Berkeley UNIX. No changes to Ctrace were necessary to accommodate any of these machines or variations of UNIX. In addition, it has been used with a cross-compiler to debug programs on 3B20 DMERT, and on the LSI-11 and MC68000 microprocessors.



---

**This page intentionally left blank**

---

## VAX11 Compatibility on PDP-11s

*Donn Seeley*

University of California at San Diego  
Department of Chemistry B-014  
La Jolla, CA 92093

8.2 is a UNIX operating system for the PDP which has filesystems that conform to the structure of the filesystems of the VAX running 4.1BSD, thus allowing a PDP to mount VAX filesystems or vice versa in a dual-ported disk or controller situation. The system consists of a modified Ritchie C compiler and C library, a 2.8BSD kernel and debugging tools, certain 2.8 utilities for the small PDP such as a small *csh* and *vi*, and as much 4.1BSD software as will run without modification, including *fsck*, *df*, *getty* and so on. Advantages of the system include rapid and trivially simple file transfer between PDP and VAX, reciprocal filesystem construction and maintenance, and the capacity to spread the availability of peripheral devices to more than one CPU (e.g. PDP files may be backed up to tape from the VAX, all printing can be channeled through one PDP, etc.). A side benefit of the particular implementation is more complete binary file compatibility between PDP and VAX.

---

## VAX11 Compatibility on PDP11s

Donn Seeley  
UCSD Chemistry Dept. RRCF

### The Problem

We have four old nonseparate PDP11s running V6 UNIX. Due to budget constraints we can't replace them with VAX 11/730s or 11/750s and we can't justify new peripherals. None of these systems has direct access to a tape drive. Of the three systems with RK05 cartridge drives only one can be easily backed up by cartridge. How can we keep these systems backed up? How can we transfer files?

### Several Solutions

- . The first solution was implemented back in the days when one PDP11 had access to a tape drive. All the systems were linked by CAMAC serial lab interfaces. A protocol was set up for these lines which allowed one system to make a request for a disk block on another system, and the other system's disk I/O system would service the request. Effectively the "block" I/O disk subsystems of the systems were transparently available to every other linked system. This allowed one to "mount" the filesystems of other machines, provided the mounting was read-only and the other machines were up and running UNIX. This enabled the operators to make dumps of any filesystem on any machine using the tape drive attached to the central machine. This solution was complicated by the advent of a new VAX 11/750 in the facility; the tape drive was moved to the VAX and the PDPs were left without tape access.
- . The second solution is the one we presently use (or suffer under). All the PDP systems are connected as before, but one of the PDPs shares a disk controller with the VAX. Both the VAX and the PDP boot from the same 300 Mb disk drive. Most user filesystems are on the booting drive; a second 300 Mb drive contains two user filesystems, two development filesystems and three large unused filesystems. For backups on the PDPs we create a V6 filesystem on one of the large unused partitions and make dumps into files on it. A program called "grab" is used to read this filesystem from the VAX and transfer the dumps to tape. Ordinary users can also use "grab" to transfer their files from one machine to another, since "grab" is built to understand the differences between the filesystem formats of PDP V6 UNIX and VAX 4BSD UNIX. This arrangement is slow and frequently unreliable, so a better situation has been devised.

- 
- . The third solution is the one we will eventually settle on (if our users ever let me schedule a flag day for the conversion!) It came from an idea that if we ran Berkeley V7 on all the machines, both the PDPs and the VAXen, then we might have a common filesystem format among all the machines and one machine could dump and restore both PDP and VAX. Also, users on a PDP could mount read-only any VAX filesystem and hence file transfer would become a trivial matter. I asked Bill Jolitz about this idea at the Santa Monica Usenix last year, however, and he said that it would be difficult to make the PDPs understand the VAX arrangement since the PDP uses a different format for 32-bit integers than the VAX.

### A Dead End

Jolitz was right in that if I were to modify the 2.8BSD kernel so that it would handle VAX-style filesystems, I would end up doing a lot of work and the new kernel would be very inefficient. The problem is that the PDP ordinarily puts the most significant 16-bit word of a 32-bit integer in the lower byte address, while the VAX puts it in the higher address. Almost every Version 7 filesystem construct uses 32-bit quantities: superblocks, freeblocks, inodes and indirect blocks. I would need to add code to the kernel that would swap the words of 32-bit integers in every operation that read in or wrote out filesystem information, and worse, all the programs which deal with the filesystem would have to be altered in the same way, even those programs which currently use the same source code for both machines. The irony is that, apart from this difference in long integers, there is essentially no difference between the two types of filesystems.

### Do PDP11s Really Have 32-bit Integers?

After a while, the thought occurred to me that the PDP11 is really a 16-bit machine, and the range of its operations on 32-bit quantities is actually rather limited.

- . The PDP has no 32-bit integer add or subtract operation. There are instead instructions which allow one to add or subtract the carry bit of a previous 16-bit add or subtract operations to or from a 16-bit word.
- . The PDP integer multiply and divide operations are in theory 32-bit oriented, but in fact they have certain problems when used in true 32-bit arithmetic. As a result the Ritchie PDP11 C compiler does 32-bit multiplication and division with subroutines. Also, multiplication and division must be done in register.

- 
- . 32-bit shift operations must also be done in register.
  - . There are no 32-bit move, test or compare instructions, except with floating point. There are a few 32-bit integer operations that can be performed by the floating point unit, but the Ritchie C compiler restricts its use of these because not every PDP11 has a floating point unit.

### Deep Compatibility

My conclusion was that provided 32-bit integers are kept in register with the high order 16-bit word in the lower numbered register, it would be possible to keep 32-bit integers in memory with the high order word in the higher address with no significant overhead either in code bulk or in efficiency. If I just changed the C compiler so that 16-bit words were swapped on every transfer to and from registers, every C program I compiled would be able to handle VAX-style long integers. More than just disk-handling programs would be affected; every program which produced 32-bit integer data that might need to be exchanged between VAX and PDP would suddenly be freed from compatibility problems. A change in one program would enable me to avoid (source-level) changes in the kernel and a host of other programs.

### The Nature of the Changes

One way to think of the compatibility effort is to consider it as a new UNIX port between two very similar machines. The first step was to develop a "cross-compiler" that would enable me to make programs on a normal 2.8 BSD system which would run on an emulation of the "target" machine. Three long evenings' work on the Ritchie C compiler's code table on an idle PDP11/34 over a weekend produced a system that would generate VAX-style code sequences for 32-bit integers in a small group of test programs I had devised. I was surprised and pleased that although there were many changes necessary to the code table entries for long integers, the rest of the compiler was only very lightly dependent on the order of the 16-bit words in a long integer; I have so far found only 3 places which require different code depending on the layout of long integers, while 2 other places have received the benefit of a portable recoding.

After I got the first version of the compiler up, I constructed a set of C library routines that served to emulate the "target" machine on my normal 2.8 system. This compatibility library basically just manipulated system calls and subroutine calls so that 32-bit arguments such as disk addresses and times could have their words appropriately swapped. I was then able to demonstrate the viability of the

---

technique by bringing up the `mkfs` filesystem-creating program and showing I could turn out filesystems that were identical to those produced by the VAX version of `mkfs`. With a little more work I was able to bring up `fsck` as well. Neither of these programs required any changes to the source, although it was unnecessary to define an alternate set of preprocessor parameters so that 2.8 BSD idiosyncracies such as short inodes and short superblocks could be avoided. Also, some of the structure definitions had to be changed slightly to account for the VAX Portable C Compiler problem of requiring 32-bit quantities to be aligned on 4-byte address boundaries. A number of code table bugs turned up at this point but with a little effort I got everything working correctly.

I was ready to tackle the kernel. To my surprise there were only a few changes needed: the long multiplication and division routines in the machine language assist needed fixing so that the return values went in the right places; the code for reading and writing 3-byte disk block addresses in inodes required trivial modifications; and the two system calls which returned long values in register, `time` and `lseek`, had to undergo word swap surgery.

### An Experiment

Very quickly I was brought to the point of trying to boot a system. I put a few evenings into fully converting the C library, spent a night getting the standalone boot in order, then cooked up a batch of standard programs like `init` and the shell. A filesystem was made to hold the root of the new UNIX and everything was installed on it using the `proto` mechanism of the `mkfs` program. I crossed my fingers and tried to boot. The first try didn't work, and I found a bug in the modified C library that I thought was responsible. I fixed it and tried again, and it worked.

### 11/40 Floating Point

More compiler debugging followed this, and eventually I was ready to try the system out on our Cal Data Products 135, a machine which emulates a PDP11/40. This posed two new problems: first, the CalData was a system that booted from a dual-ported foreign-vendor disk controller which it shared with a VAX11/750, and it had no standalone boot driver and the code in the system driver for dual-porting had never been tested; and second the compiler would have to generate 11/40 floating point instructions, since the old V6 compiler had been hacked to use them and people would expect the new compiler to handle them as well.

---

To make a long story short, adding the floating point was much more trouble than changing the word order in long integers. It was only in December that I finally got the floating in such a state that it was safe to start bringing up more utilities. The system now successfully runs almost all of the 2.8 BSD software I have tried; a few programs are still affected by mysterious compiler bugs but things like `make`, the VAX `vi` editor, the C-shell and so on ported essentially without trouble.

## Benefits

We have already realized some benefits from the new system, which we have jokingly called "8.2" ("2.8 BSD with the words swapped") Since it has been so difficult to back up the PDPs, they have received once-a-week or even once-a-month backups. The 8.2 areas of the disk, unlike the Version 6 areas, can receive much more regular coverage because they can be backed up and restored by the VAX using the normal 4.1 BSD `dump` and `restor` programs. I have already trashed my 8.2 root once while playing with the kernel and I managed to save my work even without a tape drive on the CalData because I had a fresh backup from the VAX in hand. Now all I have to do is to convince all of our users who have saved their data on Version 6 format RK cartridge disks that they won't lose anything by converting to the new format...

---

# The IS/1 Workbench for VAX/VMS

*William Torcaso*

Interactive Systems Corporation  
1212 Seventh Street  
Santa Monica, CA 90401

As the successor to the DEC PDP-11, the VAX 11/780 is obviously a suitable machine on which to implement UNIX. VMS is the VAX operating system supplied by DEC, and it clearly was influenced by UNIX. But VMS differs from UNIX in several important aspects of the file system, the process environment, and the user interface.

Since VMS was initially the only VAX "game in town," we implemented a UNIX emulator for it. This allowed us port to the VAX a large collection of UNIX tools (*sccs*, *make*, etc.) and have them run under VMS. Each successive major release of VMS has been more friendly to this effort.

This talk covers the major obstacles presented by VMS and show how we adapted to them. We also discuss the trade-offs between performance, compatibility with VMS utilities, and faithful UNIX emulation, as well as the lessons we and our many users have learned in the process of using such a UNIX emulator.



---

**This page intentionally left blank**

Chairperson: *Roger Sippl*  
Relational Database Systems

## Research Database Management Software for UNIX-based Microcomputers

*F. W. Stitt*

Clinical Data Research Services, Inc.  
340 Lombard Street  
San Francisco, CA 94133

Good biomedical database systems are hard to find and hard to design, because of the very nature of the data (i.e. noted for its diversity and complexity) which the databases need to deal with. For example, it is often necessary to store, retrieve, and analyze data from surveys, clinical trials, and so forth. Much of this data is not of a sort that is easily dealt with by the standard relational model popular today. There exists some software on large mainframes, but what is available is not entirely adequate. One often finds that the tools for data entry and validation, if available and usable at all, provide far less functionality than what's required.

A system comprising two VAX 11-750's with UNIX plus the 'Unify' database system and the BMDP statistical analysis package was described. An Onyx system is also being acquired to run the database software.

Unify was chosen over several other database packages because of the fact that it is actually a hybrid network and relational database system, and therefore better suited to the requirements. It provides efficient data access by using hashing and B-trees. It also includes an extensive C application library, and an interactive query-by-example facility. The package itself consists of something called Unitrieve (the database handler), plus a screen handler and menu processor.

The other database packages evaluated were LOGIX and Informix (purely relational), and IMP (an ISAM-based system). The speaker felt that none of the query languages with any of the four products were truly English-like (despite claims to the contrary by the vendors), though all were adequate. All provided an adequate C language interface.

It was also pointed out that Unify's hashing algorithm for record location results in excellent performance, but requires that the database be rebuilt as it grows.

---

RESEARCH  
DATA MANAGEMENT  
SOFTWARE  
FOR UNIX-BASED  
MICROCOMPUTERS

Frank W Stitt MD  
San Francisco California USA

The software requirements of data management are discussed, notably the complexity and diversity of research data, with some possible approaches to the problem. Statistical packages, data management and database management systems can be interfaced together, and to report generators and tabulation programs.

A fully interfaced system for research studies using a newer approach, which includes a microcomputer-based system for data entry, data management, and database management has been implemented: the system uses a 16-bit microcomputer with a hard disk and tape backup.

The operating system is UNIX, developed at Bell Laboratories, which provides a productive environment for software development or for the support of software packages.

Keywords: Microcomputers; database management systems; UNIX; statistics; biomedical research

## 1. Introduction

The ever-increasing demands for more information on health-related problems have placed a great burden on those responsible for the collection, organization and reporting of the data required to answer these questions.

Biomedical data is noted for its diversity and complexity.

Existing packaged software is oriented towards statistical analysis and hypothesis testing, whereas in biomedical studies the main effort is required for data entry, data validity checking, data reorganization, tabular reporting and hypothesis generation.

While researchers have good and well-tested software available for statistical analysis there is a paucity of reliable, conversational, user-oriented data and database management software, required for the bulk of the research effort.

This paper will discuss the use of microcomputers to replace data-processing which could only previously be handled on large mainframe computers.

---

## 2. Previous experience with an interfaced system

The author (Stitt 1978, 1979, 1981) has previously described the assembly of a data management and analysis system which can supply easy-to-use, conversational tools for the management and analysis of a diversity of biomedical research studies, including surveys, registries, cohort studies and clinical trials.

The software modules chosen are characterized by English-like control languages and by interfaces between modules.

The system can be used by doctors and nurses as well as computer programmers and statisticians.

It is currently installed on a DEC VAX 11/750 computer.

Many of the elements of a research data management system can be "downloaded" to 16-bit microcomputers.

## 3. Application software for 16-bit microcomputers

### 3.1. The IMP data management system

IMP is a simple yet powerful data entry and data management system written in the "C" programming language for UNIX-based systems.

IMP allows a form to be described and formatted on the screen and can be used with "dumb" or intelligent terminals. Data is then entered from the keyboard and verified field-by-field; any errors are displayed at the terminal and can be immediately corrected.

Records are indexed using up to 10 keys; this means that the records (or cases) can be retrieved sorted or one at a time based upon any combination of the keys. Over forty utilities are available for searching, listing, data reorganization or report generation.

IMP is a powerful and flexible front end for clinical trial and occupational health data processing, where the data must be entered into the computer for statistical analysis and cross-tabulation.

### 3.2. The INFORMIX Relational Database Management System

INFORMIX's components include an interactive query language, and interactive data entry and maintenance program, a report writer, audit trail and recovery programs, and all the utilities needed to create, modify, and optimize databases.

This set of tools is flexible enough to allow many simple applications to be simply configured, and not programmed at all. For applications where detailed customization is needed, interfaces to high-level programming languages are provided, such as C or CIS-COBOL.

INFORMIX uses the relational data model which is renowned for its elegance and simplicity. The relationships between data groups do not have to be

---

rigidly defined in advance, and a large number of variables and files can easily be maintained within the database, based upon the creation of a data dictionary used by all the other programs within the system. Additional variables can be added or indexes created at any time, interactively.

The flexibility of a relational database system lends itself to many health-care oriented database applications, such as tumor registries and occupational health monitoring.

A disadvantage of relational databases is the requirement to create temporary files after a join of two files, based on a common key (unless the software permits virtual files). This can cause problems in a strongly update-oriented environment where all the files for a single case are to be viewed and updated at once. A hierarchical or network database, with the ability to store implicit relationships with "pointers", may have some advantages for administrative databases.

INFORMIX is strongly user-oriented, with a simple structure, an excellent manual, and a simple but powerful report writer.

A recent addition to the suite of programs available in the INFORMIX system is the PERFORM screen generation utility. PERFORM allows the painting of screens using a full-screen editor (such as "vi"), dynamic relational joins for cross-linking of files, and query-by-example retrievals. For data entry, PERFORM allows range, list and table validation, with defaults by field, including dates. Entries may be forced or verified.

PERFORM is a useful tool for data entry, validation, or retrieval.

### 3.3. UNIFY

UNIFY is a high-performance database development tool for UNIX programmers. It has several important features:

- 1) The system can bypass the normal UNIX file-handling mechanism, and write directly to disk.
- 2) It allows explicit relationships to be described with "pointers", and is therefore uses a hybrid relational-network model.
- 3) It uses hashing algorithms as well as B-trees: this greatly improves update times for prime keys, while allowing partial matches on secondary keys.
- 4) It has some powerful development aids, including: a menu-handler, screen-formatter, and the most extensive C-language application library, with 90 C routines. It also has an interface to RM-COBOL and to FORTRAN-77.
- 5) UNIFY is strongly integrated, and requires few keystrokes to use.
- 6) It has a very useful query-by-forms (or example) utility, with the ability to create and drive "canned" reports to the screen or the printer by interfacing the query-by-forms selection to the report

---

writer. For many unsophisticated users this is all they will need.

- 7) A full implementation of SEQUEL (the IBM relational data language) is expected in the next release.
- 8) UNIFY has excellent security protection, down to the field level. Each user can have his own view of the database defined by the database administrator.

Some disadvantages are: a lack of a database load feature; the report writer requires some programming, and is not as non-procedural as INFORMIX. With the present release (2.0) no level break processing or headers are available.

### 3.4. LOGIX

This relational DBMS is closely integrated into UNIX: all commands appear to the user as UNIX utilities, and are invoked from the command level; it uses UNIX files rather than an internal format, making it easy to interface with other systems.

It appears that the system may be wasteful of disk space, because every operation creates a new relation, and the history of each file is stored with the data. There does not appear to be a recovery procedure from tape, but good privacy and security is possible. The system is strongly oriented to data management.

### 3.5. SEQUITUR

This DBMS allows processing of variable-length records of text. It has a unique text-editor built into the system, allowing large databases of text to be handled internally. It also has a Query-by-Example language allowing easy retrievals and updates by the user, and also permits virtual files resulting from relation joins (thus avoiding the creation of large temporary files).

## 4. Evaluation of DBMS's for UNIX-based microcomputers

An evaluation has been made of three major DBMS which will run on 16- and 32-bit microcomputers with UNIX, and a data management system. These evaluations will be found in the Appendix "DBMS Evaluations". The criteria were selected from a variety of sources in the literature and from personal communications, and reflect the personal biases of the author, although guided by the work of Francis (1981) in statistical software evaluation.

The ratings are graded from + = Least desirable, to ++++ = Excellent; a "!" means that the system is outstanding on this criterion, probably because it was designed to optimize the feature. "?" means that the feature cannot be evaluated from the documentation, and "-" means that the feature is absent. For the section on vendor support, a "\*" indicates that the author has used the system in a production environment.

IMP is not graded in this group because it is not commercially available from UC Berkeley at present, although it has extensive use within the

---

university (as has UNIX itself).

## 5. Conclusions

Although most large biomedical research databases are being managed with ad hoc study-by-study programs, or with statistical packages, increasing use is being made of database management systems.

Since these DBMS's have no statistical capability, interfaces must be constructed between the DBMS and the statistical system to be used for the analysis.

Our experience has shown that these interfaces can be constructed, and there is increasing experience of this approach now available. Some data management systems have been specifically designed with this goal in mind, and more can be expected.

Other computer aids for forms development and production, data dictionaries, text encoding with thesauri and table generation have also been found to be most useful, and can be recommended.

There is at present a lack of good available software for inter-variable consistency checking, and for the production of case-wise reports from complex hierarchical databases; it is to be hoped that progress will be made in these directions soon by workers active in this field.

Exciting new developments are taking place in microcomputer systems, the newer of which will support software development or packaged applications which would previously have required machines three-to-five times the cost.

A significant factor in the utility of these machines is the availability of the UNIX operating system, with its associated utilities and programming languages. Powerful database management systems are now available for a variety of biomedical research endeavors.

## References

Francis I. (1981), Statistical Software: A Comparative Review. New York: Elsevier North-Holland, Inc.

Stitt F.W. (1978), Clinical Research Data Management and Analysis - a Physician's View". Drug Information Journal: April, 98-110.

Stitt F.W. (1979), Software for Data Management and Analysis of Large-scale Biomedical Studies. in "Techniques for the Processing or Analysis of Large Data Sets". Manila: Proceedings of the 42nd Session of the International Statistical Institute.

Stitt F.W. (1981), Clinical Database Management: a Clinician's View, in "Concepts Related to Database Management". Toronto: Canadian Organization for the Advancement of Computers in Health. Fifth Conference on the Health Professional and the Computer.

# APPENDIX: DBMS EVALUATION

| FEATURE                                                  | UNIFY      | LOGIX | IMP  | INFORMIX   |
|----------------------------------------------------------|------------|-------|------|------------|
| MODEL:                                                   | RN         | R     | I    | R          |
| (R = Relational; I = ISAM;<br>N = Network )              |            |       |      |            |
| 1.0 DATA MANIPULATION:                                   |            |       |      |            |
| 1.1 Data manipulation processes                          | +          | ?     | -    | +          |
| 1.1 Privacy/security techniques                          | ++++       | ++++  | ++++ | -          |
| 1.3 Disaster recovery procedures                         | ++++       | ++    | ++   | +          |
| 1.4 Data integrity controls                              | ?          | ++++  | ?    | ?          |
| 1.5 Format modification ability                          | +          | ++    | +++  | +          |
| 1.6 Redundancy/consolidation control                     | ++++       | ++    | ++   | ?          |
| 1.7 File growth                                          | ++         | ++++  | +++  | +++        |
| 2.0 QUERY LANGUAGE:                                      |            |       |      |            |
| 2.1 Availability of feature                              | Y          | Y     | N    | Y          |
| 2.2 Ease of use                                          | +++        | +++   | -    | ++         |
| 2.3 Capability                                           | ++         | ++    | -    | ++         |
| 2.4 Query-by-forms (example)                             | ++++       | ?     | -    | +++        |
| 2.5 SQL data language                                    | Y          | -     | -    | -          |
| 3.0 APPLICATION PROGRAMMING COMPLEXITIES:                |            |       |      |            |
| 3.1 Program/data independence                            | +++        | +++   | +    | +++        |
| 3.2 Methods used to define manipulation<br>and retrieval | ++++       | ++++  | ++   | ++++       |
| 3.3 Subsystem view development                           | -          | -     | -    | +          |
| 3.4 Schema description process                           | +++        | -     | +    | +++        |
| 3.5 Programmer skill requirement                         | +++        | ++++  | +    | ++         |
| 3.6 C language interface, N of routines                  | 89         | ?     | 40   | 11         |
| 3.7 Other language support - RM/COBOL                    | Y          | N     | N    | N          |
| - CIS COBOL                                              | N          | N     | N    | Y          |
| - FORTRAN                                                | Y          | N     | N    | N          |
| 4.0 PHYSICAL FILE DESIGN:                                |            |       |      |            |
| 4.1 File organization                                    | ++         | +++   | ++   | ++         |
| 4.2 Access method: (H)ash; (B)trees                      | HB         | B     | B+   | B+         |
| 4.3 Indexing methods                                     | +++++      | ++++  | ++   | ++++       |
| N of secondary indexes                                   | N fields 7 |       | 8    | N of field |
| 4.4 Record types supported                               | ++++       | ++++  | +    | ++++       |
| (date, time, money etc.)                                 |            |       |      |            |
| 4.5 Record change capability                             | +++        | ++++  | +    | +++        |
| 4.6 Ability to combine records                           | +++        | ++++  | +    | +++        |
| 4.7 Virtual fields                                       | ++++       | +     | -    | -          |
| 4.8 File space management                                | ++         | -     | +    | -          |
| 4.9 Logical record definition                            | -          | -     | ?    | -          |
| 4.10 Logical structures                                  | +++        | ++++  | -    | ++++       |
| 4.11 Explicit relationships                              | ++++       | ?     | -    | ++         |
| 5.0 SYSTEM INSTALLATION:                                 |            |       |      |            |



|                                                              |          |       |       |         |
|--------------------------------------------------------------|----------|-------|-------|---------|
| 5.1 Physical file distribution control                       | ?        | +++   | -     | -       |
| 5.2 Database loading facility                                | -        | ++++  | ++++  | ++      |
| 5.3 Hardware requirements<br>(++++ = Major; + = Minimal)     | +        | ++++  | +     | ++      |
| 6.0 DBMS UTILITIES:                                          |          |       |       |         |
| 6.1 Performance statistics gathering                         | -        | ++    | -     | -       |
| 6.2 Database statistics                                      | ++       | ?     | -     | -       |
| 6.2 Minimum reorganization for<br>performance                | +++      | ++    | -     | -       |
| 6.3 Data dictionary                                          | +++      | +     | +++   | ++      |
| 7.0 SECONDARY FEATURES:                                      |          |       |       |         |
| 7.1 System performance                                       | ++++!    | ?     | ++++  | ++      |
| 7.2 DBMS maintenance policy<br>(O = OEM; U = End-user sales) | O++      | U+    | -     | U++     |
| 7.3 Systems design and development time                      | ++       | +++   | ++    | ++++!   |
| 7.4 System designer training time                            | ++       | +++   | ++    | ++++    |
| 7.5 Ease of installation                                     | +++      | ++++  | +     | ++++    |
| 7.6 Documentation                                            | ++       | +++   | +     | ++++!   |
| 7.7 Vendor support                                           | +++ (*)  | ?     | -(*)  | +++ (*) |
| 7.8 Vendor responsiveness to change                          | ?        | ?     | -     | ++      |
| 7.9 Customer experience (N installations)                    | ++       | ?     | +     | +++     |
| 8.0 COMPATIBILITY WITH UNIX:                                 |          |       |       |         |
| 8.1 Integration into operating system                        | ++++     | ++++! | ++++  | ++++    |
| 8.2 Integration with file system                             | ++++     | ++++! | ++++! | ++      |
| 8.3 Accessibility with C programs                            | +        | ++    | +++   | +       |
| 9.0 COMPATIBILITY WITH DISTRIBUTED<br>PROCESSING:            |          |       |       |         |
| 9.1 Modular back-end                                         | ++       | ?     | -     | -       |
| 9.2 Concurrent access                                        | ++       | +++   | +     | ++      |
| 9.3 Audit trails                                             | ++++     | +     | +++   | ++      |
| 10.0 SYSTEM DESIGN:                                          |          |       |       |         |
| 10.1 Integration of features                                 | ++++     | ++    | +     | ++      |
| 10.2 Data dictionary support of UNIX files                   | -        | ++++  | ++++  | -       |
| 10.3 Schema entry                                            | ++       | ++    | +     | +++     |
| 10.4 Ad-hoc creation of logical structures                   | -        | +++   | -     | +       |
| 10.5 Variable length and nonexistent fields                  | ++++     | ++++  | -     | -       |
| 10.6 System Limits                                           |          |       |       |         |
| Files/database                                               | 100      |       | 1     | 30      |
| Fields/file                                                  |          | 128   |       |         |
| Records/file                                                 |          | 1M    | 32K   |         |
| Fields/database                                              | 1300     |       |       | 300     |
| bytes/record                                                 | 25,600   |       | 16M   | 2048    |
| Files/join                                                   | No Limit | 2     | 2     | 2       |
| 11.0 EFFICIENCY AND PERFORMANCE:                             |          |       |       |         |
| 11.1 Bias in favor of update vs retrieval                    | U++++    | UR    | U+++  | R++     |
| 11.2 Performance implications of concurrency                 | ++       | ?     | ++    | -       |
| 11.3 No reorganization                                       | +++      | +++   | ++++  | +       |

---

## 12.0 MAINTENANCE:

|                                          |      |      |      |      |
|------------------------------------------|------|------|------|------|
| 12.1 Security of administrator functions | ++++ | +++  | ++++ | -    |
| 12.2 Backup to UNIX files and tape       | ++++ | ++   | +++  | ++   |
| 12.3 Fine tuning requirements            | ++++ | +++  | ?    | ?    |
| 12.4 Maintenance while system is running | ++++ | ++++ | ++   | +    |
| 12.5 No skills required                  | +++  | ++++ | +    | ++++ |

## 13.0 APPLICATIONS:

|                                          |       |      |       |      |
|------------------------------------------|-------|------|-------|------|
| 13.1 Application software development(*) | ++++! | ?    | +     | +++  |
| 13.2 On-line update                      | ++++  | ++++ | ++++! | ++   |
| 13.3 Alias support                       | +++   | -    | -     | -    |
| 13.4 "Canned" requests                   | ++++  | ++++ | -     | +    |
| 13.5 Development aids                    |       |      |       |      |
| Menu handler                             | ++++  | -    | -     | +    |
| Screen formatter                         | ++++  | -    | ++++! | +++  |
| Internal controls                        | ++++  | ?    | ?     | ?    |
| 13.6 Report writer                       | ++    | +    | +     | ++++ |

## 14.0 DATA ENTRY &

### VALIDITY CHECKING:

|                           |      |      |       |     |
|---------------------------|------|------|-------|-----|
| 14.1 Data entry           | ++++ | +    | ++++! | ++  |
| 14.2 Implicit data types  | +++  | ++++ | ++++! | +   |
| 14.3 Cross-field checking | ++   | ?    | ++    | -   |
| 14.4 Table lookup         | +    | -    | ++++  | +++ |
| 14.5 Field defaults       | -    | -    | +++   | +++ |

---

**This page intentionally left blank**

---

# **The Design and Implementation of the DB Relational Database Management System**

*J. Robert Ward*

International Institute For Applied Systems Analysis  
Schlossplatz 1  
A-2361 Laxenburg  
Austria

The speaker described the design philosophy of a database system which currently runs on the VAX and the 11/70, and which will soon run on one or more 68000-based machines. The basic design goals were: (1) flexibility, and (2) that it be efficient for query processing.

The system provides a query language with syntax and typing rules similar to those of C. A user can specify the format of output in a printf-like way, to facilitate coexistence with other UNIX programs, such as *awk* and *sed*. It's also easy to combine this database system with existing report generators.

The query processor does its work by parsing a query, and converting the parse tree to a pipeline of action-specific nodes. Each node in the pipeline contains an internal representation of a selection constraint, and a pointer to a coroutine which acts on tuples to perform the required selection operation. This approach permits the selection procedure to run without the need for temporary disk files. The coroutines themselves are simulated by recursive function calls.

---

# The Design And Implementation Of The DB Relational Database Management System

*J. Robert Ward*

International Institute For Applied Systems Analysis  
Schlossplatz 1  
A-2361 Laxenburg  
Austria

## ABSTRACT

The DB relational database management system is a series of programs written to provide a general tool for computer users at IIASA. The system is currently implemented on a VAX-11/780 and a PDP-11/70‡, both running under the UNIX† operating system.

The DB system was designed to be extremely flexible and efficient in terms of processing. It is well integrated into the working environment presented by UNIX.

This paper discusses the design and implementation of the DB system. In particular, it describes the method of data storage and how a powerful pipe-line algorithm aids the query processor.

## 1. Introduction.

The DB relational database management system was written at IIASA in order to provide a general tool for our computer users. This paper discusses the overall design and implementation of the system.

The system had two primary design objectives. First, it had to be flexible : it should be able to meet a wide range of applications. Second, the system had to be reasonably efficient in terms of processing.

This paper describes how the goal of flexibility has been attained by integrating the system into the existing framework of UNIX [RITC74] as far as possible. The choice of a powerful query language has also been highly important.

There are two major aspects of the design of the DB system that help it to evaluate queries efficiently. First, a reliable and fast set of access method procedures facilitate the query processor in retrieving stored data from disc. Second, a powerful pipe-line algorithm enables the query processor to evaluate relational expressions without having to store temporary results on disc.

---

‡PDP and VAX are Trademarks of Digital Equipment Corporation.

†UNIX is a Trademark of Bell Laboratories.

---

Some of the features normally found in large database systems are omitted from the DB programs. For instance, DB does not support user defined views on relations, it is not a multi-user system and it is geared towards query processing rather than the update of stored data. Nevertheless, the system has been applied to a diverse range of data handling problems, some of which are described below.

The DB programs are currently implemented on a PDP-11/70 and a VAX-11/780, both running under the UNIX operating system. They are written in portable C [KERN78], except for a few lines of assembler code in the VAX version.

The actual usage of the DB system to construct and maintain a relational database is detailed in the users' manual [WARD82].

## 2. Integration Within UNIX

Unlike some DBMS's, notably INGRES [STON76], the DB system does not impose a new working environment on the user. DB was developed to be a standard computing tool and can be combined with other UNIX utilities. For instance, the output from the query processor can be piped to standard programs such as *awk* or *sed*. This aspect immediately increases the system's flexibility since it easily allows one to combine the DB system with existing programs.

The absence of a specialised environment also allows one to develop a command pre-processor that interfaces between the user and the program. Such an interface can hide details from naive users. It can also combine the DBMS with report generators or other specialised programs.

There is no "hidden place" where data is stored. Nor is there any concept of a "system catalogue". All information concerning a single relation is contained within one binary file ‡. This means it is possible to use standard UNIX commands such as *cp*, *chmod* or *tar* on individual relations, without having to learn new commands. For instance, a user may delete a relation simply by using the standard *rm* command. There is no need to provide a special command that would have to delete the stored data and then ensure that the relation's entry in a global system catalogue is also removed. Again, this approach simplifies the implementation and the effort involved in learning the system.

### 2.1. The DB Programs

The DB system is broken up into logically distinct programs. The various programs are briefly described here -

**Dbcreate** reads ASCII data and converts it to the special file format required by the DB programs. This is generally the first step in setting up a database.

**Dbappend** is used to place additional ASCII data into a relation.

**Dbis** reports information about the data stored in a database. It is the DB equivalent of the standard UNIX *ls* command.

**Db** is the main query processor. Db may be instructed to list data from a database or to combine or transform existing data to produce new output.

**Dbedx** invokes a screen editor [PEAR80], allowing the user to update a relation interactively.

**Dbmodify** "cleans up" a relation by reformatting it and freeing unused disc space. This program can also construct a secondary index for a relation.

---

‡One could argue that DB should store data in ASCII files in order to fulfill the integration into UNIX. This would, however, limit the system's efficiency.

---

### 3. Command Language

A retrieval command to the query processor is phrased in a language based on Codd's relational algebra [CODD72]. The generality of the language allows both simple and complicated queries. This is important since it allows the user to formulate casual one-off queries easily. On the other hand, it is possible to develop a query instructing DB to evaluate a complicated relational expression.

Designers of relational DBMS's seem to have concentrated on the relational calculus rather than the algebra. Relational programming languages based on the calculus include SQL [ASTR76] and QUEL [STON75]. The predominance of such systems may reflect users' needs for transaction processing rather than query processing and Date [DATE80] mentions that the calculus is easier to optimise. I, however, agree with King [KING79] who states that the algebra is more adept to developing complicated queries. The DB system allows for complicated query processing on stored data, rather than the update of data. Therefore, the algebra was chosen as the basis for the query language.

DB uses punctuation characters to signify relational operators. The syntax of these operators is described in [WARD82]. For example, the query -

**Books :: author == "Austen"**

means 'list tuples from relation *Books* such that fields from domain *author* have the value "Austen"'. Similarly, this query -

**Books \*\* Loans**

means 'list the join of relations *Books* and *Loans*.' (In this case, the join is over domains having the same names in both source relations.)

Besides Codd's basic relational operators, DB includes an aggregation operator. This is a generalised operator by which one can find maxima, minima, totals, etc. within a relation. It allows one to split up a relation into groups of tuples, each group having the same value in one or more domains, and to perform some summarising operation on each group. (This is similar to the operation of 'glumping' described in [HITC77].)

DB also provides a sort operator. This is useful to order a large listing of a relation. For instance, this query -

**Books ## author title**

means 'sort relation *Books* over domains *author* and *title*.'

The query language allows one to evaluate relational expressions and assign them to new output files. The output file may be created in relational binary format or as an ASCII file. For instance, this query evaluates the relational difference of relations *Books* and *Loans* and places the output in a new relation file called *Result* -

**Result <= Books - - Loans**

DB also allows one to call upon the standard C language procedure *printf* to format listed output. For instance, the following query lists the output fields according to the given string -

**Result "Author = %s, title = %20s, bookno = %d\n" <== Books - - Loans**

---

### 3.1. Scalar Expressions

The query language allows one to perform limited operations on the data extracted from a set of relations. The syntax of the query language dealing with the manipulation of scalar quantities is based upon that of the C language.

Each domain of a relation has a specific type. The DB programs support storage of integer data, floating point data and character strings. The VAX implementation also allows a domain to hold packed decimal values. Fields of differing types may be combined or compared : type rules analogous to those of the C language then apply.

DB also provides a few built-in scalar procedures. Some of these routines allow one to manipulate string fields in a limited fashion, or to match string fields according to a regular expression, in the manner of the standard UNIX program *ed*. Other routines allow one to manipulate integers representing calendar dates, or to print such quantities in a variety of formats.

### 3.2. Pre-Processor Input

The query input is sent through the pre-processor of the C compiler before being parsed and executed. The pre-processor expands defined macro statements and generally extends the system's flexibility.

## 4. Implementation Of Stored Data

The DB programs share a library of access method procedures. These routines are responsible for storing and retrieving data. The procedures themselves are not described here (they are documented in [WARD82] ). Instead, I shall describe how relation files are formatted and the sorting and paging algorithms.

In the interests of portability and simplicity, the access methods use the UNIX file system. Although not optimal, storing data in standard files facilitates the implementation and the overall integration within UNIX .

### 4.1. Storage Of Data On Disc

All information concerning a single relation is contained within one binary file. A relation file stores information describing the relation itself and the data of the relation. It may also store secondary index information. Such a file has a layout as shown in fig. 1. The head of the file contains information about the relation itself. The remainder of the file holds the data.

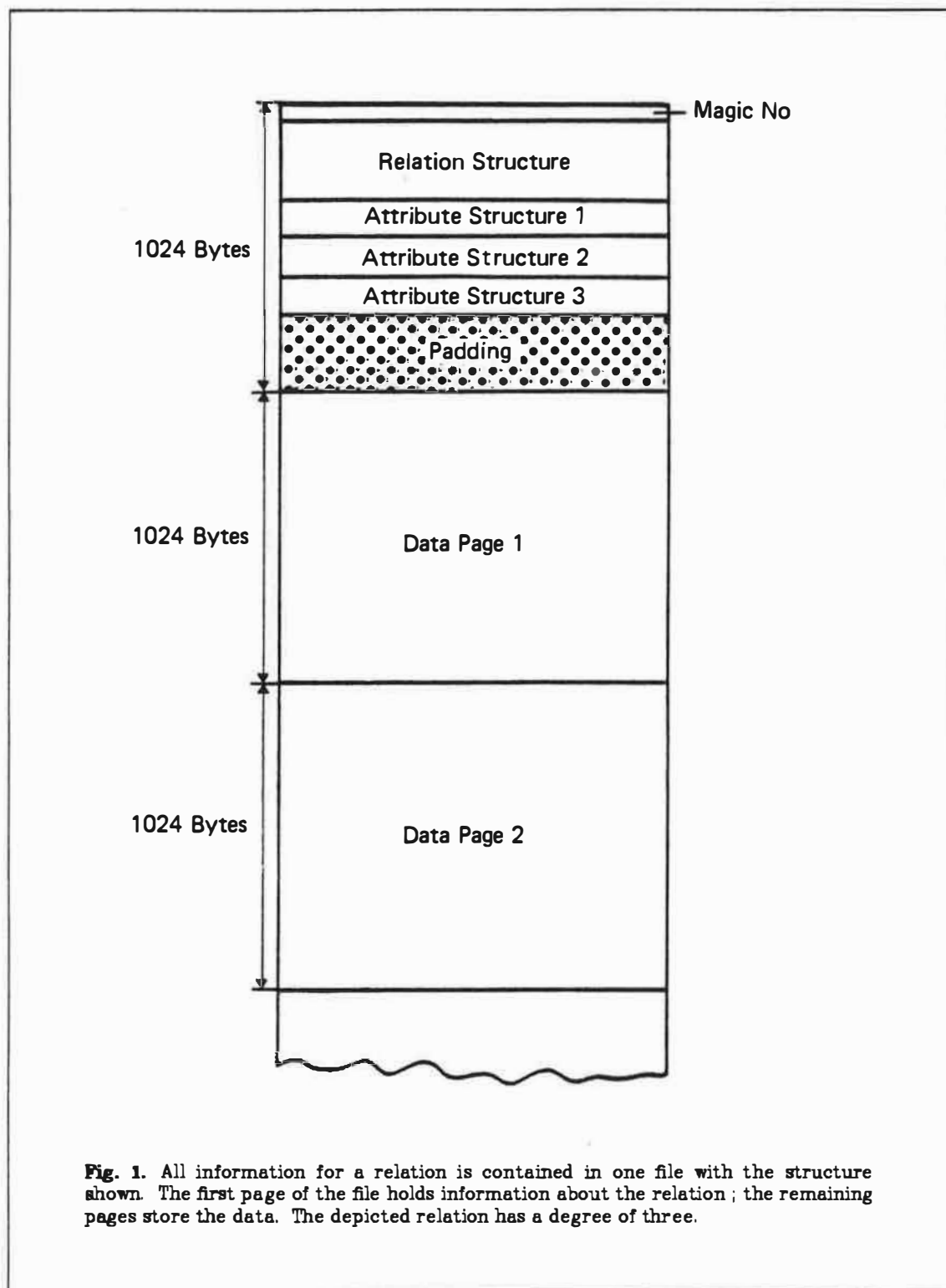
### 4.2. Stored Information About A Relation

At the head of each relation are two bytes containing a 'magic' number. This number identifies the file as being a relation file. When the relation is opened, these two bytes are checked to ensure that the file is a valid relation.

Following this magic number is a *relation* structure. This contains all information concerning the relation as a whole. For instance, this structure contains the relation's cardinality, its degree, its storage mode, etc.

After this structure come a series of *attribute* structures. An attribute holds information about one domain. For instance, it holds the domain's name, its data type, etc.





**Fig. 1.** All information for a relation is contained in one file with the structure shown. The first page of the file holds information about the relation ; the remaining pages store the data. The depicted relation has a degree of three.

### 4.3. Data Pages

After the information about the relation, the file is padded up to the next block boundary. The remainder of the file is devoted to the actual data.

The data portion of the file is divided into *pages*. A page is the unit of I/O. Each page occupies 1024 bytes, this being a multiple of the operating system's disc block size, and is aligned on a page boundary.

DB organises pages in much the same way as does INGRES. Both systems have a common method of placing tuples into pages and of linking pages on disc. This algorithm is described in [STON76]. The two systems, therefore, share the same advantages and deficiencies in this respect.

### 4.4. Storage Modes

The DB programs support three different storage modes : heap, sort and hash. These storage modes reflect how tuples are ordered inside a relation and therefore how keyed tuples may be quickly retrieved. The DB programs can optimise access to specific tuples provided that an appropriate storage mode has been chosen.

Tuples within a *heap* relation are contained in a random order. It is not possible to optimise access to such a relation and the entire file must be read whenever it is accessed. This mode is useful for small relations or for those that are often updated or accessed infrequently. It is also used for building temporary relations.

The tuples within a *sorted* relation are contained in a specific order. Keyed tuples can be quickly located through a binary search. This mode is the default for the DB programs and generally the most useful.

The DB programs also support *hashed* relations where each data page acts as a hash bucket.

### 4.5. Secondary Indices

One may use the program *dbmodify* to form a secondary index for a relation. A relation may have any number of secondary indices. The DB program chooses an appropriate secondary index, if keyed access through the primary relation is impossible.

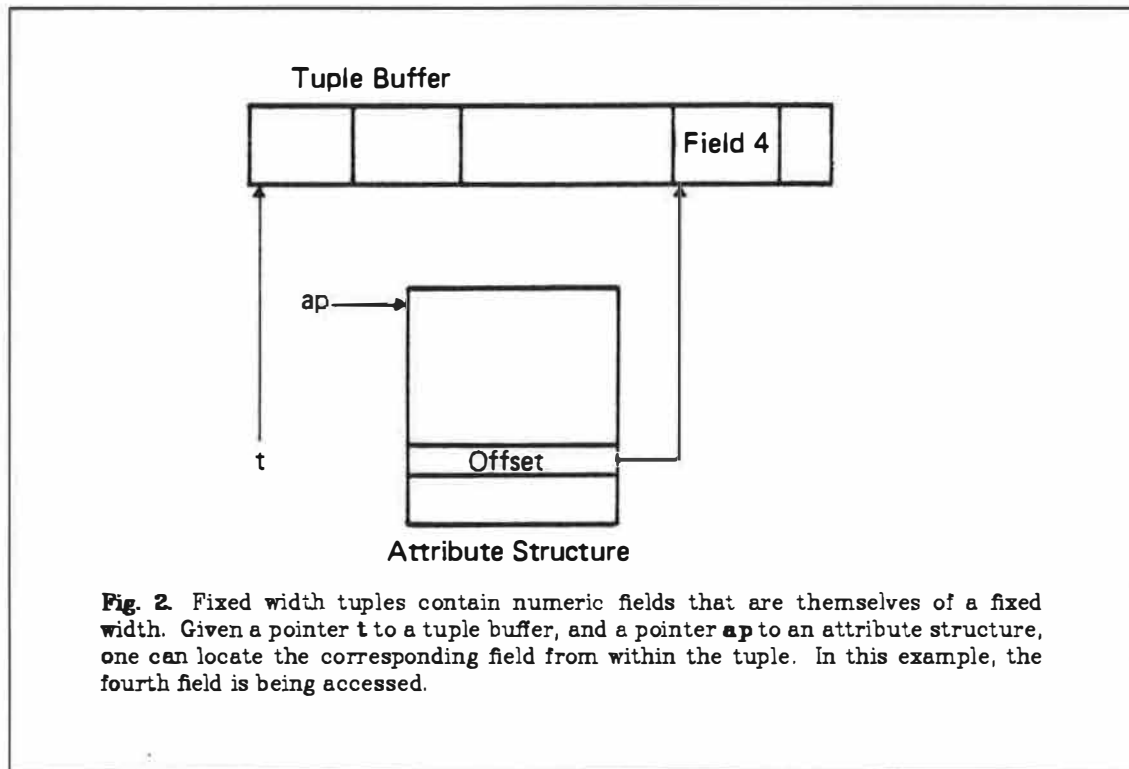
A secondary index is itself a relation consisting of *tuple identifiers* that reference tuples within the primary relation. The access methods ensure that all secondary indices are updated whenever an addition is made to the primary index.

The relational information concerning a secondary index is stored on the first disc block, after that for the primary index. Should there be insufficient room, this information is placed at the end of the file. Secondary index tuples are stored on data pages at the end of the file.

### 4.6. Representation Of Tuples

One basic restriction is imposed by the way in which a relation is divided into pages. No tuple may span across a page boundary, thus limiting the maximum size of any tuple.

Tuples are stored in two different formats. If a tuple contains only numeric data, each field occupies a fixed width and the size of the entire tuple is also a fixed quantity. To access a field from such a tuple, one need only to know the address of the tuple and the offset to the field. This offset is contained in the corresponding attribute structure (see fig. 2).



Character strings are held in fields of variable width. The alternative, storing strings in fixed width fields, would waste disc storage. It would also mean that one would have to specify the expected maximum length of such fields.

The total size of such a tuple is also variable : the first two bytes hold its total length. A field containing fixed width data is referenced as before, by an offset found in the corresponding attribute structure. For character string fields, however, this offset references two bytes inside the tuple that specify a further indirection to where the data itself is located. The data is placed at the end of the tuple (see fig. 3).

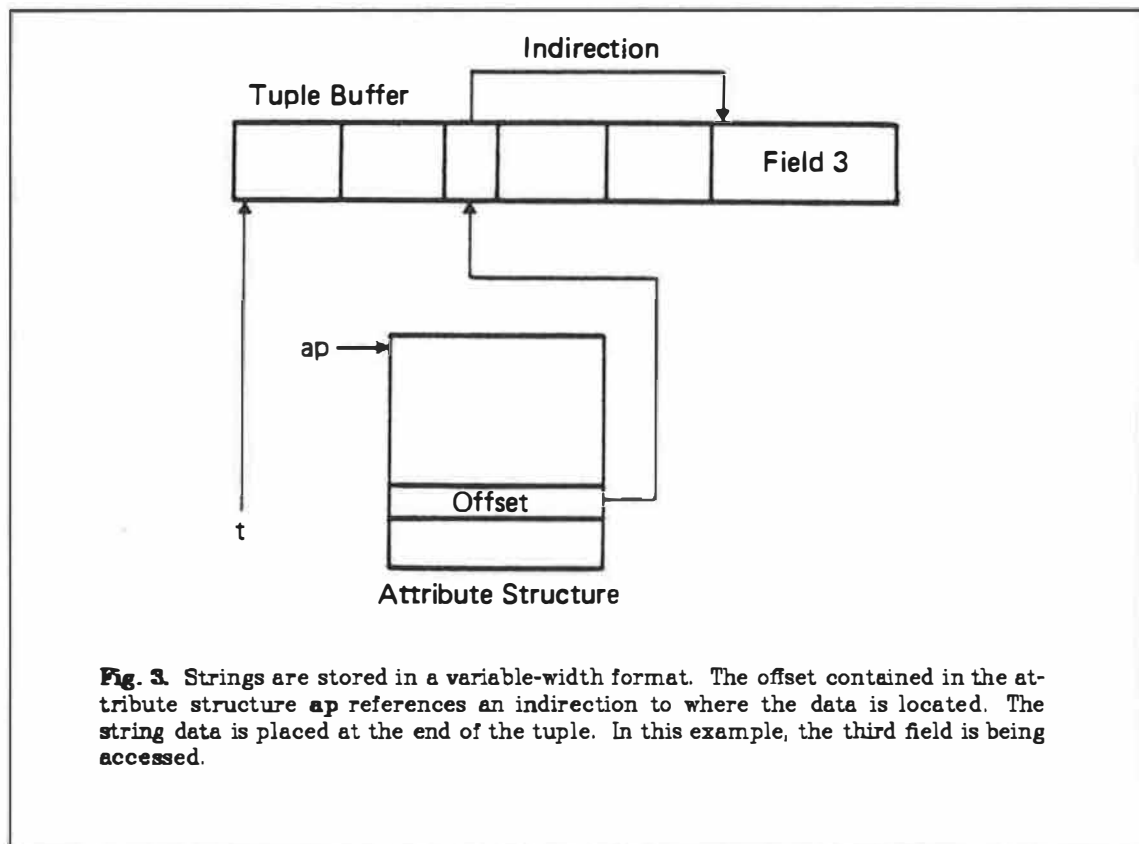
#### 4.7. Concurrency Control

Relations are locked while being updated. The locking mechanism is implemented by creating a file that acts as a semaphore. Once a process has created this file, it has exclusive access to a small global table. The table is checked by the DB programs to determine if an open request can be satisfied. The semaphore file is then deleted and other processes can proceed in the same manner.

#### 4.8. The Paging Algorithm

In order to minimise the number of disc read and write operations, the access methods employ a 'least recently used' paging algorithm. When a tuple is referenced, a search is made though a buffer pool to see whether it is already in core. If not, a buffer is made available and the appropriate page obtained from disc.

It is necessary to differentiate between those pages that may be required again and those that must be allowed to migrate quickly to disc. Pages are



**Fig. 3.** Strings are stored in a variable-width format. The offset contained in the attribute structure **ap** references an indirection to where the data is located. The string data is placed at the end of the tuple. In this example, the third field is being accessed.

usually not written until the cache is completely allocated and a buffer is needed for a new page to be read in. There are a few exceptions to this rule. Before a relation is closed, any write-pending pages are flushed. Secondly, the DB programs support the output of relational data through a UNIX pipe. Since random access I/O is impossible, pages must be written as soon as they become full and they are not preserved in the buffers. Finally, new pages belonging to a heap relation are not preserved since they are unlikely to be required again. This prevents write-pending pages from occupying the cache at the expense of others that may be required soon.

#### 4.9. The Sorting Algorithm

The DB programs rely heavily on a fast algorithm to sort a relation. This routine is called when a relation is to be restructured by `dbmodify`. It is also called from the query processor when evaluating relational expressions.

Additional main storage is allocated into which the tuples are read. They are sorted in core by an optimised quicksort algorithm and then written out to a temporary file. This process may be repeated depending on the size of the relation to be sorted and the availability of memory space. The temporary runs are then merged to form a single sorted relation.

Because of the large virtual address space on the VAX, the sorting algorithm often finishes without having to go into a merge phase. Sorting a relation is, therefore, correspondingly faster on the VAX implementation.

---

#### 4.10. Overview Of The Access Methods

The overall performance of the access methods seems to be quite reasonable and these procedures permit fast insertion and deletion of tuples. Keyed tuples may be located quickly, provided the relation has few overflow pages. The variable width tuple format is of particular benefit for storing character strings as it allows considerable economy of disc storage.

The major problem with the existing system is that it becomes slow when much data is appended to a sorted or hashed relation. Additional tuples are placed on overflow pages and these tend to degrade the access methods' efficiency.

The access methods do not preserve the order of tuples within a chain of overflow pages. An entire chain must be searched on a keyed access and this can be an expensive operation. The program *dbmodify* is provided to help with this problem. This reorganises the relation by copying the data to a new file consisting only of primary pages.

If the length of an overflow chain exceeds the allocated number of internal page buffers, a search may be prohibitive and will cause the paging algorithm to thrash. A chain is searched starting from a primary page and so reading through the overflow pages will cause the paging algorithm to discard those read at the head of the chain.

More importantly, it is not obvious to a user why the system should become so slow when updating sorted or hashed relations. One may convert the relation's storage mode to a heap beforehand and restore it afterwards. However, this is clearly a hindrance and something that should be taken care of by the system itself.

Both these problems could be overcome by the use of a B-tree storage method [KNUT73], although at the expense of additional disc requirements. A B-tree ensures that the relation's structure remains balanced and it preserves the exact order of sorted tuples. This would largely obviate the need for the *dbmodify* program, since there would be no need for the relation to be periodically cleaned up. I agree with [STON80] on the benefits of this method. The preservation of the sorting order would also benefit the query processor, a point to which I shall return.

The fact that a tuple is not allowed to span a page boundary enforces a limit on the maximum size of any tuple. One can foresee applications where one might wish to store large quantities of text, perhaps several paragraphs, in a single field. This restriction makes such an application impossible.

The DB programs provide no integrity control of their own and UNIX has no form of crash recovery. This is a serious deficiency.

Although reasonably portable, the access methods rely on the low-level system calls to the operating system. UNIX itself does not give much support for database management systems. In particular, some form of concurrency control would be of appreciable benefit: a parameter to the *open* system call should be available, by which one can request exclusive access to a file. Some enhancement to the file system in general would be welcome: a form of asynchronous I/O would make considerable difference to the overall efficiency.

---

## 5. Implementation Of The Query Processor

This section explains the implementation of the program *db*, the main query processor.

A powerful pipe-line algorithm ensures that queries are processed efficiently. The algorithm avoids writing temporary results out to an intermediate disc file. The pipe-line is a recursive structure set up after parse time.

### 5.1. Parsing The Query Input

The parser was written with the aid of the YACC compiler-compiler [JOHN75]. The parser accepts the command language and builds an in-core representation of the query. This is a tree structure containing only the names of domains and relations, the values of scalar constants, etc.

The leaves of the parse tree correspond to source relations. Nodes within the tree correspond to relational operators.

### 5.2. Construction Of The Pipe-Line

The next stage is to transform the basic tree formed by the parser into the *pipe-line* structure. There is a one-to-one mapping from the nodes of the parse tree to those in the pipe-line. Every relational operator appearing in the query corresponds to a unique node in the pipe. Thus the pipe may be a single line or a tree.

The nodes of the pipe contain more information than those in the parse tree. Each node holds, amongst other things, a description of the relation at that stage of the pipe.

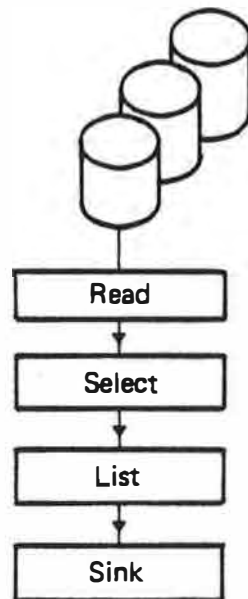
A recursive procedure is called upon to build the pipe-line. The leaves are the first to be transformed into pipe nodes and this involves opening the source relations.

This transformation procedure is also responsible for initialising the pipe. For example, it may be called upon to build a node corresponding to a relational selection. It then constructs an internal representation of the selection constraint. Fig. 4 shows an example of a pipe-line corresponding to the selection and subsequent listing of tuples read from a source relation.

A further example may be seen in the case of the join operator. Here, the transformation routine determines the commonly named domains of the source relations. (The join operation is defined as being over those domains from the source relations that share the same names.) It initialises the node with information showing how the source tuples are to be compared and of how the output tuples are to be constructed. Finally, it forms a description of the resulting relation.

One final operation is applied to the pipe. If the operator at the base of the pipe is not an explicit assignment to an output file, it is assumed that the relational expression is to be evaluated and listed to the standard output. An extra node corresponding to a print operation is then built into the root.

Thus the construction of the pipe-line involves opening the source relations, identifying the format of the relation at each node and generally initialising internal data structures.



**Fig. 4.** This pipe-line corresponds to a relational expression of the form `Books :: author == "Austen"`, that is, to a selection followed by a listing.

### 5.3. Processing Tuples

Once the pipe-line is constructed, it is used to evaluate the specified relation expression. Apart from the information described above, each node of the pipe holds the address of a procedure to be applied to the data stored in that node. Each such procedure behaves as a *coroutine*. (The notion of a coroutine is discussed in [KNUT77]). Every implemented relational operator has a corresponding coroutine.

Most coroutines may be thought of as consumers and producers of tuples. A coroutine is generally activated by its *parent* coroutine. The parent requests one tuple from its *child* and then hangs, waiting for the request to be satisfied. A coroutine always has one parent but may have zero, one or two children, depending whether it corresponds to source relation or to a unary or binary relational operator.

The coroutines are simulated by recursive function calls. Any coroutine can call upon a primitive operation to request a tuple from a child: it can then transfer a tuple back to its parent through a return statement. The pipe structure preserves static information for each coroutine while it is inactive.

The coroutine at the base of the pipe is the first to be invoked. This coroutine requests a tuple from its child. The child in turn requests a tuple from its child, and so on down the pipe. Eventually, a coroutine at the other end of the

---

pipe is called upon to deliver a tuple. This coroutine activates the paging mechanism and retrieves a single tuple from disc. The tuple is then passed back to its parent.

Once a coroutine has obtained a tuple from its child, it applies whatever operation is necessary. For instance, the coroutine corresponding to a projection transforms the fields within the tuple. It then returns the transformed tuple to its parent. A selection coroutine keeps requesting tuples from its child until finding one that matches the specified constraint. The tuple is then handed to the parent coroutine.

The tuple is thus successively transformed and manipulated until it reaches the base of the pipe-line. The coroutine at the base is a data sink : it discards the tuple and immediately requests another from its child. Thus the pipe is kept in motion until there are no more tuples to be processed.

The overall effect can be viewed as a flow of tuples down the pipe, while requests for tuples travel up the pipe.

#### **5.4. Listing And Assignment Of Relational Expressions**

Just as for the other relational operations, a coroutine is used to print tuples. This coroutine requests a tuple from its child and waits. When it receives a tuple, it prints it and then hands the tuple to its parent. A similar coroutine is used to assign tuples to an output relation.

This method allows one to preserve an intermediate result of the pipe-line if so desired.

#### **5.5. Implementation Of The Join, Intersection And Difference Operators**

The join, intersection and difference operator are implemented by having the pipe-line sort all received tuples into order before they are passed to the appropriate coroutines. It is then trivial to merge the sorted tuples. It has been shown in [BLAS77] that such a strategy is generally efficient.

During construction of the pipe, two sort coroutines are inserted between the operator node and its children. When activated, a sort coroutine reads all available tuples from its child, sorts them and places the result into a temporary file. It then hands back one tuple at a time to the operator coroutine. Fig. 5 shows the formation of a pipe-line corresponding to a listing of a join of two source relations.

The same sort package used to reformat a relation may also be instructed to read tuples from a coroutine. Although sorting becomes a bottle-neck for processing these operators, it is generally preferable to locate matching tuples through a random access search or by making multiple passes over the data.

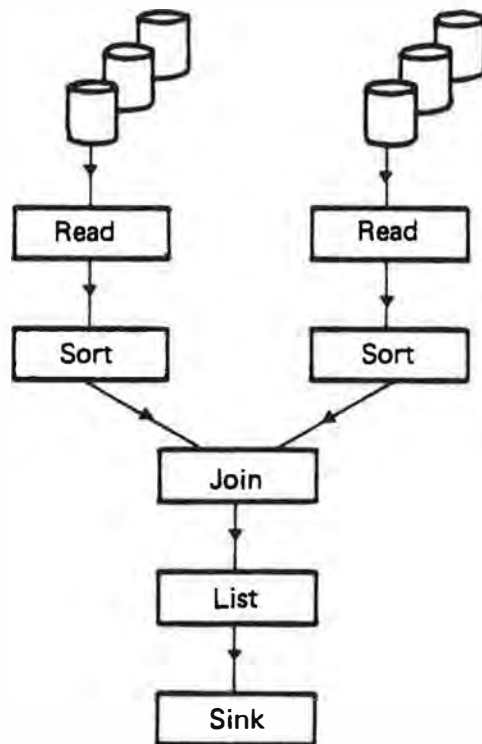
The fact that the sort package places its output into a real file aids the implementation of the join operator. The join coroutine may need to rescan one of its operands. It is easy to back up the scan when reading from a real relation file : it would be more difficult if the coroutine were reading directly from a child. If a rescan is necessary, it is likely that the tuples are still available in the buffer pool.

#### **5.6. Advantages Of The Pipe-Line Algorithm**

The pipe-line algorithm allows an uninterrupted flow of tuples between successive processing steps. Generally, it avoids intermediate results having to be written to temporary disc files.

Another advantage is the ease with which algorithms can be coded to





**Fig. 5.** This pipe-line corresponds to a relational expression of the form **Books \*\* Loans**, that is, to a listing of a join of two source relations. The join coroutine must receive tuples in sorted order : therefore sorting coroutines are inserted into the pipe.

perform the various relational operations. The coroutine procedures themselves are very small and efficient and each performs a well defined unit of work. The coroutines are written so that they are independent of one another. A coroutine requests a tuple via a primitive operation. No coroutine need know from where that tuple has come or to which parent it will be returned.

Generally, the coroutines transfer pointers to tuple buffers between themselves. Only very rarely must an entire tuple be copied. The cost of activating a coroutine is small and so this algorithm requires little overhead. Indeed, it has been shown in [STOR77] that the cost of passing control between pipe-line nodes is not significant compared with other costs of a DBMS.

The join, difference and intersection coroutines each need to receive tuples in sorted order. Sorting, therefore, becomes a bottle-neck throughout processing. However, the bottle-neck is localised : a highly efficient sort package makes considerable difference to the overall performance. Profiling indicates that a third of the time taken to process an "average" query is spent in sorting. Coding

part of the sort routine in assembler language would probably prove beneficial to the system's efficiency.

Since the coroutines are independent, there is no fundamental limit on the complexity of the pipe structure. Such a limit is instead imposed by the operating system. The number of source relations cannot exceed the permitted maximum number of open files. A complicated relational expression may, therefore, have to be split up into smaller processing steps.

In order for this algorithm to function, there must be sufficient main storage to hold the pipe-line and the coroutine procedures. Space for holding page buffers and for sorting is also required. All this can be accommodated on a PDP-11/70 without trouble, although it does require the split I and D space facility of this machine.

### 5.7. Further Improvements

Sorting is a bottle-neck and, although the sort package is reasonably fast, DB should recognise if it can be avoided.

On the present implementation, DB resorts a relation even if it is already correctly ordered. Clearly, it should exploit any existing order in the source relations. Again, a B-tree storage mode would be beneficial. Such a mode enables tuples to be stored so that their order is preserved during updates.

Most coroutines can be written so that they preserve the order of tuples flowing in the pipe. DB should optimise construction of the pipe to take advantage of this fact. It should eliminate and combine sorts wherever possible, as discussed in [HALL76] and [SMIT75].

## 6. Benchmark Tests

The following figures give an indication of the efficiency of the DB query processor.

Three benchmark tests were conducted on a VAX-11/780 that was otherwise only lightly loaded. In each case, the test consisted of evaluating the join of two source relations and placing the output into a new relation file. The tests differed in the size of the source relations. In each case, most of the source tuples from one operand matched those from the other.

| Cardinality  |              |        | Recorded Times |           |
|--------------|--------------|--------|----------------|-----------|
| 1st Relation | 2nd Relation | Output | CPU            | Elapsed   |
| 1050         | 1155         | 1050   | 9.5 secs       | 26.0 secs |
| 41135        | 41125        | 41135  | 7.7 mins       | 19.4 mins |
| 52395        | 240627       | 306484 | 38.1 mins      | 97.9 mins |

Little notice should be taken of the figures themselves, except to note that the CPU and elapsed times are approximately proportional to the cardinality of the output relation. (These values cannot be extrapolated to predict the time required to process other joins: this depends on the size of the input tuples and how they match one another.)

---

## 7. History And Comparisons

DB was not conceived as a complete DBMS until the project was well under way. Until then the only available DBMS running under UNIX had been INGRES. Although INGRES is a valuable research tool, it suffered from several deficiencies making it unsuitable for our proposed applications. First, it was then only available on Version 6 UNIX ; the integration of INGRES within UNIX was thought to be poor and this limited its potential applications ; it was also slow and made heavy demands on the system.

Much of the code was overlaid between four or five concurrent processes. There was a high cost in swapping these processes and in copying data from one process to another. All these processes shared the same access methods library and this was found to be inefficient.

Initially I began work on changing the access methods code in the hope that this would lead to an eventual overall improvement. This approach was eventually abandoned and work commenced on a new DBMS, that is, on DB. Therefore the DB access methods code stems from that of INGRES, while query processing is solved in a completely different way. INGRES uses 'decomposition' to process queries [WONG76] ; DB relies on the pipe-line algorithm.

The technique of pipe-lining has already been introduced in data-processing systems, for instance [HOUS79] , although the exact means of implementation varies widely. The CODD system [KING79] introduced coroutines to transfer control between pipe-line nodes. In turn, CODD was developed from an earlier system called PRTV [TODD76] .

CODD uses an entirely different method to implement coroutines [MOOD80] . At run-time, each coroutine possesses a separate stack : a switch between coroutine activations involves access to the stack pointer and other information that can be obtained only through a knowledge of the language implementation.

In DB, coroutines are invoked through recursive calls to procedures and the data for each coroutine is preserved within the pipe-line. In CODD, however, this information is preserved on a coroutine's stack. This implies that the implementor must know the space needed for each stack in advance. This approach does make coding the algorithms somewhat easier since one does not have to differentiate between variables that are preserved across coroutine activations and those that are truly 'local'. On the other hand, it leads to a wastage of memory space since all variables are preserved on a coroutine stack, even if this is unnecessary.†

## 8. Current Applications

At the time of writing, DB has been fully operational for almost eighteen months. It is being used to manage several databases at IIASA.

One application has been implemented for the Personnel Department at IIASA. This database is used to maintain information concerning employees at the Institute. Since most people who maintain this database are unfamiliar with the UNIX system, a C Shell command processor [PEAR82] is used to interface between the DBMS and the users. This command processor hides many details of the database implementation from those who maintain the data. The command processor combines DB with report generators and other utilities. The database is updated frequently by using the *dbedx* screen editor to change or

---

†It should be mentioned that CODD allows a generalisation of the pipe-line structure unobtainable with DB : the pipe need not be a tree structure, but can be a directed graph.

add new information.

Another application has been formed for the Resources And Environment Task at IIASA [FEDR83]. A database has been constructed to store experimental data. This database currently has a size of over 1 megabyte and contains relations with a cardinality of over 80000 tuples. The typical queries presented to DB are not complex but involve processing joins over such relations. The database is eventually expected to grow to a size of perhaps 10 megabytes.

A third example of its current use at IIASA is an application for the Budget And Finance Department [GODW82]. This department uses DB to look after the Institute's bookkeeping. DB maintains a database of monetary transactions on a set of accounts. It is used to calculate the tax returns on these accounts and to perform various totaling operations on the data. For this application the cardinality of the input data is small (approximately 2000 new tuples per month). However, the query developed for the tax calculations and for evaluating the movements on each account is complicated, involving over eighty relational operations. Despite the small quantity of input data, DB may perform over ten million tuple processing operations. Because of the financial nature of the task, it is essential that all arithmetic processing be accurate. Therefore packed decimal arithmetic is used throughout.

## 9. Conclusion

The DB system is currently being used to maintain several diverse databases, each with its own special requirements. The choice of the relational algebra as the basis of the query language allows one to formulate both simple and complex queries. This fact, coupled with its overall integration into UNIX has facilitated construction of these databases.

I have shown how a powerful pipe-line algorithm can aid a DBMS in processing queries. Besides avoiding unnecessary storage of intermediate results on temporary disc files, this approach simplifies the task of coding algorithms to implement the various relational operations.

Future enhancements could well improve the overall performance. The implementation of a B-tree storage method and automatic optimisation of the pipe-line would benefit the query processor.

Any data processing system can be ultimately judged by how well it can be applied to solving real problems. Despite its limitations, the DB system seems to have been largely successful in this respect.

## 10. References

- ASTR76    *Astrahan, M. M. et al., System R: A Relational Approach To Database Management, ACM Transactions On Database Systems, Vol. 1, No. 2, pp. 97-137, (June 1976).*
- BLAS77    *Blasgen, M. W. and Eswaran, K. P., Storage And Access In Relational Databases, IBM Systems Journal, No. 4, pp. 363-377, (December 1977).*
- CODD72    *Codd, E. F., Relational Completeness Of Database Sublanguages, Database Systems, Courant Computer Science Symposia Series, Vol. 6., Prentice Hall, Englewood Cliffs, New Jersey, (1972).*
- DATE80    *Date, C. J., An Introduction To Database Systems, Second Edition, Addison-Wesley Publishing Co., Reading, Massachusetts, (1980).*
- FEDR83    *Fedra, K., Maatta, R. and Ward, J. R., Using A Database Management System In An Environmental Study - An Application Example, IIASA, Schlossplatz 1, A-2361 Laxenburg, Austria, (forthcoming).*

- GODW82 Godwin-Toby, W., The Inhouse Operation Of IIASA's Bookkeeping Using The Relational Database Program DB. (IIASA internal paper), *IIASA, Schlossplatz 1, A-2361 Laxenburg, Austria*, (1982).
- HALL76 Hall, P. A. V., Optimisation Of Single Expressions In A Relational Database System, *IBM Journal Of Research And Development*, Vol. 20, No. 3, pp. 244-257, (1976).
- HITC77 Hitchcock, P., User Extensions To The Peterlee Relational Test Vehicle, *Proceedings Of The 2nd International Conference On Very Large Databases*, pp. 169-180 : North-Holland Publishing Co., (1977).
- HOUS79 Housel, B. C., Pipelining : A Technique For Implementing Data Restructurers, *ACM Transactions On Database Systems*, Vol. 4, No. 4, pp. 470-492, (December 1979).
- JOHN75 Johnson, S., YACC - Yet Another Compiler-Compiler, *Computer Science Technical Report No. 32, Bell Telephone Laboratories, Murray Hill, New Jersey*, (July 1975).
- KERN78 Kernighan, B. W. and Ritchie, D. M., The C Programming Language, *Prentice Hall, Englewood Cliffs, New Jersey*, (1978).
- KING79 King, T. J., The Design Of A Relational Database Management System For Historical Records, *Doctoral Dissertation, Corpus Christi College, University Of Cambridge, Cambridge, England*, (November 1979).
- KNUT73 Knuth, D., The Art Of Computer Programming, Vol 3., pp. 473-479, *Addison-Wesley Publishing Co., Reading, Massachusetts*, (1973).
- KNUT77 Knuth, D., The Art Of Computer Programming, Second Edition, Vol 1., pp. 190-196, *Addison-Wesley Publishing Co., Reading, Massachusetts*, (1977).
- MOOD80 Moody, K and Richards, M., A Coroutine Mechanism For BCPL, *Software, Practice And Experience*, Vol. 10, pp. 765-771, (1980).
- PEAR80 Pearson, M. M. L., Using The Computer To Communicate : An Introduction To Text Processing At IIASA - The Edx And Nroff Programs. (Working Paper WP-80-111), *IIASA, Schlossplatz 1, A-2361 Laxenburg, Austria*, (July 1980).
- PEAR82 Pearson, M. M. L., Users Guide To Datactr And Namebank (IIASA internal paper), *IIASA, Schlossplatz 1, A-2361 Laxenburg, Austria*, (November 1982).
- RITC74 Ritchie, D. M. and Thompson, K., The UNIX Time-Sharing System, *CACM*, Vol. 17, No. 7, pp. 365-375, (July 1974).
- SMIT75 Smith, J. M. and Chang, P. Y. T., Optimising The Performance Of A Relational Algebra Database Interface, *CACM*, Vol. 18, No. 10, pp. 568-579, (October 1975).
- STON75 Stonebraker, M., Getting Started In INGRES - A Tutorial, *Electronics Research Laboratory Memorandum ERL-M518, University Of California, Berkeley*, (April 1975).
- STON76 Stonebraker, M., Wong, E., and Kreps, P., The Design And Implementation Of INGRES, *ACM Transactions On Database Systems*, Vol. 1, No. 3, pp. 189-222, (September 1976).
- STON80 Stonebraker, M., Retrospection On A Database System, *ACM Transactions On Database Systems*, Vol. 5, No. 2, pp. 225-240, (June 1980).
- STOR77 Storey, R. A. and Todd, S. J. P., Performance Analysis Of Large Systems, *Software, Practice And Experience*, Vol. 7, No. 3, pp. 363-369, (June 1977).
- TODD76 Todd, S. J. P., The Peterlee Relational Test Vehicle - A System Overview, *IBM Systems Journal*, Vol. 15, No. 4, pp. 285-308, (1976).
- WARD82 Ward, J. R., An Introduction To The DB Relational Database Management System, Second Edition, *IIASA Software Library Series LS-7, IIASA, Schlossplatz 1, A-2361 Laxenburg, Austria*, (December 1982).
- WONG76 Wong, E. and Youssefi, K., Decomposition - A Strategy For Query Processing, *ACM Transactions On Database Systems*, Vol. 1, No. 3, pp. 223-241, (September 1976).

---

# Interactive System/Three and the Intel Data Base Processor

*John R. Levine*

INTERACTIVE Systems Corporation  
P.O. Box 349  
Cambridge, MA 02238

The Intel 86/440 is a microcomputer-based back-end relational database machine. A host computer stores data into and retrieves data from a database by communicating with the iDBP, making use of the iDBP's set of database primitives. (The term 'primitive' may be a bit misleading, though, since database operations such as 'join' and 'select' are provided directly by the iDBP. In other words, some of the 'primitives' are rather powerful.)

The iDBP permits a host system to define, select, retrieve, update, and delete data. It provides access control on both a record and an item level. It also provides all necessary concurrency control, to ensure the integrity of the data stored in iDBP databases. It supports both structured and unstructured data, and relationships between structured and unstructured files. Features such as transaction logging are also provided.

What is necessary to make effective use of the iDBP is an appropriate software package in the host machine(s), which provides an interface between users and the iDBP itself, and which translates commands and queries into equivalent host→iDBP message packets which execute the iDBP database primitives.

The speaker described a set of tools running under IS/3 (Interactive's version of UNIX System III), which permit a user to utilize iDBP functions. For example, there is an interactive inquiry language, 'ddml', which permits a user to enter iDBP commands and immediately see the response. There is also a precompiler which processes programs written in an 'extended' version of C (consisting of normal C plus embedded iDBP interface language) and produces a pure C program (intended to be system-independent) which can then be compiled and run to interact with databases in the iDBP.

In the work-in-progress category are other programs, designed more for the casual user (i.e. non-programmer). For example, there is a screen-oriented query language, modeled after IBM's Query-by-Example language.

---

## INTERACTIVE System/Three and the Intel Data Base Processor

*John R. Levine\**

INTERACTIVE Systems Corporation

### ABSTRACT

The Intel Data Base Processor is an intelligent back-end data management system. This paper describes the iDBP and how it is integrated into the INTERACTIVE IS/3 system.

## 1. OVERVIEW

The Intel Data Base Processor, also known as the iDBP or the Intel 86/440, is a microcomputer based back-end relational data-base machine. It provides data-base functions for one or more hosts running INTERACTIVE System/Three (IS/3, a functional superset of UNIX† System III). All communication with the iDBP takes place through the IS/3 hosts; users do not, in general, have to deal with the iDBP directly.

The iDBP provides data-base management in a manner that is faster, more reliable, and less expensive than the equivalent capability on a host system. Unlike data-base systems that run directly on UNIX hosts, the iDBP can handle data bases with hundreds of millions of characters, and also can handle text effectively. Furthermore, it is designed to work well in an environment with many simultaneous users on several host systems, and to store and update data reliably even in the face of hardware failures.

The iDBP interface package is composed of many parts that work together to let the IS/3 user utilize the facilities of the iDBP. Both low-level and user-oriented facilities are provided.

## 2. THE iDBP

### 2.1 Features

The facilities of the iDBP include:

- Data definition, selection, retrieval, update, and deletion.
- Comprehensive access control, down to the item level.
- Integrity control of all items stored, including enforcement of inter-item and inter-file relationships.
- Concurrency control.
- Transaction logging, recovery, and restart.
- File and data-base back-up and recovery.
- Performance tuning.
- Integrated data dictionary.
- Parameterized commands and macro support.
- Full text storage, searching, and modification.
- Relationships among structured and unstructured files.
- Linked list support for hierarchical and network data bases.

---

\* Author's address: INTERACTIVE Systems Corp., P.O. Box 349, Cambridge MA 02238.  
Network addresses: decvax!yale-col:jrl, research!ima!johnl, cbosgd!ima!johnl.

† UNIX is a trademark of Bell Laboratories.

---

## 2.2 Structured and unstructured files

The iDBP stores data in *structured* and *unstructured* files. In structured files, each record is constructed of a number of *items*. Each item has a name and a type, with the types being signed and unsigned numbers of various lengths, short strings of text, and pointers to records and strings in other files. The items in each record type are known by name or position, so that requests such as "select from file X records where item Y is greater than Z" can be made directly.

Unstructured files are a simple stream of any kind of data, usually of text, and are most conveniently dealt with via string pointers from structured files. Structured and unstructured files can be logically *connected* so that large chunks of text can be handled in the same manner as structured data.

## 2.3 Views and relationships

The iDBP is designed to support primarily relational data bases. It directly provides the relational *join*, *project*, and *select* operations. It also provides a *connect* operator to define relationships between structured files (those that contain short strings, numbers, and pointers) and unstructured files (those that contain text or other strings of information) and an *order* operator to control the order in which information is presented to the host system. A *union* operator allows several similarly structured files to be treated as one.

All data is manipulated through *views*. A view is like a file except that its contents are determined dynamically from the contents of the files upon which it is based. For example, one might have a file of customers and a view of customers that owe more than \$1,000. Every time the view is examined, the iDBP automatically selects the appropriate entries from the underlying file. All of the operations described above can be used to create views. A view can be used once and thrown away, or else stored permanently. A user or a programmer manipulates views exactly the same way as files.

## 2.4 Linked lists and performance enhancement utilities

The iDBP includes several features to improve data-base processing performance. B-tree or hashed indices can be defined on any item or pair of items in a file. The iDBP automatically maintains these indices as the file is modified. The iDBP takes advantage of such indices whenever data are requested from the file or a view derived from that file. Creating or deleting an index is transparent to the user of the file. All existing views automatically use new indices.

The iDBP also has support for record pointer items to provide a "fast path" between two files or for those situations where a hierarchical or network data base is appropriate. The can link and delink records and automatically follow pointers to find all the records in a chain. The relational operations can still be used on files that are involved in linked lists, thus allowing a single set of files to be treated both as a relational data base and as a network or hierarchical data base.

## 2.5 Access and concurrency control

Every file and view has separate passwords for read access, write access, and administrative access. Views can have different access protection from the files on which they are based. This allows extremely fine-grained access protections to be easily defined. For example, a personnel file would be highly protected because it contains sensitive information. A view of that personnel file that contained just names and telephone numbers might be generally readable to use as an on-line telephone book. Note that the view contains no data of its own but just reflects the data in the underlying file, so that changes made to the names and telephone numbers in the personnel file are immediately reflected in the telephone book view.

Parameterized macros can also be defined. These macros have separate passwords for reading, writing, and execution, so execute-only macros provide another level of protection.



---

The iDBP provides full state-of-the-art concurrency control. In the event of a power loss or other system failure it can always recover the data bases back to a consistent state. File and data-base back-ups and roll-forward logs help to recover from media failures.

A group of iDBP commands can be grouped into a *frame*. The iDBP guarantees that either all of the changes in the frame will be applied or else none of them will. If the frame is blocked because records it is using are locked, the iDBP can automatically back the frame out and restart it. Users can also put commands in the frame that check various logical conditions in the data base and abort the frame if problems are discovered.

### 3. BASE INTERFACE BETWEEN THE iDBP AND THE HOST

The logical connection between the and a user program looks much like a bidirectional UNIX pipe. The user program sends a group of commands (a *request module*) to the iDBP and the iDBP sends back a group of responses (a *response module*). The iDBP commands in a module are quite flexible and can contain logical tests, *if-then-else*, loops, etc. The commands can either be compiled into the host program (as would be typical for a canned application) or else made up dynamically at run time.

Separate users have separate iDBP *sessions*. The sessions are managed by the iDBP so that a user's passwords, privileges, open files, and other context information are unique to his session. The separate sessions on the iDBP are analogous to separate processes on the IS/3 host.

#### 3.1 General structure of the interface

The user deals with the iDBP either directly through the *ddml* inquiry language or else through a C program with iDBP code embedded therein. There is a single iDBP command language, no matter whether the commands are embedded in a program, read from a file, or typed directly at the terminal.

#### 3.2 Components of the interface

**3.2.1 Precompiler.** The precompiler translates a convenient user language into the binary codes actually required by the iDBP. The user language is embedded in a host language (which is always C at this point) so that the user has the full power of the host language at his disposal when dealing with the iDBP.

**3.2.2 User interface routines.** The user interface routines support the code produced by the precompiler. They handle the run-time details of transferring information to and from the iDBP. The calls to these routines are almost always generated automatically by the precompiler.

**3.2.3 Dynamic compiler.** The user cannot always predict in advance what commands will be needed when dealing with the iDBP. To support the situation where iDBP commands are created in the running program, the dynamic compiler is provided.

**3.2.4 Interactive inquiry language.** The interactive inquiry language *ddml* lets the user type iDBP commands directly and immediately see the response. It is useful for debugging iDBP commands and for initializing and examining the state of iDBP files and data bases.

**3.2.5 System device driver.** The system device driver handles the low-level work of attaching the iDBP to the host, routing commands from user programs to the iDBP, routine responses from the iDBP back to the appropriate user programs, and performing other related housekeeping tasks.

#### 3.3 Use of the embedded language

The iDBP interface language is embedded in host language (always C at this point) programs. The precompiler reads the program with the embedded interface language and writes a pure host language program. This program can then be compiled and run like any other programs in the host language. (The UNIX utilities *yacc* and *lex*<sup>1</sup> work in much the same way.) The host code

produced by the precompiler is intended to be machine independent so that a program that is precompiled on one system can be compiled and run on another, and so that the precompiler and programs that depend on it can be moved from one host to another with minimum effort.

Each block to be treated by the precompiler is delimited by special brackets:

```
... C code ...
$[
    precompiler code
]$
... C code ...
```

The precompiler translates three types of information, *record declarations*, *commands*, and *response commands*. Each block contains only one type of information. The precompiler can tell from context the type of each block.

**3.3.1 Record declarations.** Most iDBP files will have records with many items in them. Rather than force the user to explicitly declare the disposition of each item for every record sent to and from the iDBP, the user can write a *record* declaration. The record declaration looks and acts much like a C *struct* declaration.

```
record {
    int amount;
    char *name;
    char rank[10];
    sid description;
} footype;
```

**Figure 1.** Sample record declaration

Figure 1 declares a record type *footype* with four fields. The first is a two-byte integer, then a string, another string, and finally a string-pointer item. This gets translated into, among other things, a C *struct typedef* declaration so that the user can declare variables of type *footype*. The important difference between a *record* declaration and a regular *struct* declaration is that the *record* declaration can also be used in iDBP *fetch* and *store* commands. The interface language will take care of transferring each of the items in the record in the correct order and will also do any translations needed to convert from host format to iDBP format (the byte order for longs, for example, differs greatly from one model of computer to another).

Only a limited set of types can be used in *record* declarations, namely *int*, *long*, *char*, character pointers and arrays, and special *RID* (record pointer) and *SID* (string pointer) structures. Each of these corresponds directly to an iDBP data type. For the character pointers and arrays, the precompiler adds a length item to the generated structure that is filled in on input and believed (if it is non-negative) on output. When reading, items that are too long are silently truncated to fit. All string items are null-terminated if there is room in the array. Character pointer items have space for the strings automatically allocated by the standard *malloc()* and *free()* routines.

The record declaration compiles into three things, a structure declaration, a read routine, and a write routine. The read and write routines are automatically used to help transfer data to and from the iDBP.

---

1. S. C. Johnson and M. E. Lesk. UNIX Time-Sharing System: Language Development Tools, *Bell Sys. Tech. J.*, 57(6):2155-75 (1978).

---

**3.3.2 Commands.** The precompiler understands the complete command language of the iDBP. When a the running C program flows through a block of commands, the commands are sent to the iDBP as a command module.

When data is sent to the iDBP via a `store` command, the user can specify either the explicit items or else a record type and a record instance. This allows the precompiler to pass the correct sizes and types of information to the iDBP automatically. When explicit item specifications are given, items can be copied from other records in the iDBP or can be specified by host language expressions.

**3.3.3 Response commands.** The iDBP sends back a variety of different information in response to user commands. The responses fall into two general categories, various kinds of status information and data from a data base. A given group of response commands has the structure of a list of possible responses types and what to do about them. Any response that is not listed can, at the user's option, be ignored, produce a warning, or else provoke a fatal error. A response block can, at the user's option, process a single response or else repetitively process all of the responses from an entire module.

Data is stored away in much the same way that it is sent. A `fetch` response gives a record type and a record instance to be filled in, and can specify a variable into which the cursor number is stored.

### 3.4 Dynamic compiler

The dynamic compiler is essentially the same as the precompiler except that it is invoked as the program is running. The dynamic compiler resembles in concept some similar features in IBM's System R.<sup>2</sup>

The user calls `dynopen` which opens a pipe to the dynamic compiler and returns a FILE pointer. The user then writes the iDBP commands to that file as ASCII text using the usual standard I/O library routines such as `fprintf`, `fputs`, and so forth. He calls `dyneom()` to mark the end of the module, which is then immediately compiled and passed to the iDBP. Precompiled modules can also be executed while the dynamic compiler is active.

### 3.5 Interactive inquiry language

The interactive inquiry language *ddml* is a combination of the dynamic precompiler and a response decoding program. All input is taken to be iDBP commands so it is immediately compiled and the modules are passed to the iDBP. The input language is exactly the same as for the precompiler except that source text need not be bracketed in `$[` and `]$`, and there are a few special control commands.

When *ddml* is invoked with no arguments, commands are read from the standard input. If arguments are given, they are taken to be files of commands. In that case, *ddml* automatically creates an application session, compiles and passes all the commands in the files to the iDBP, and then deletes the application session. This mode is particularly useful for executing canned scripts that define or clean up iDBP files, views, and data bases.

### 3.6 Example of use of embedded commands

Figure 2 shows a small program using embedded commands. First, the record type must be defined. A single record type can be and usually will be used in many `fetch` and `store` statements.

Second, an instance of the record type is defined. (A C `typedef` declaration is provided for each record type.)

---

2. Donald D. Chamberlin. Support for Repetitive Transactions and Ad Hoc Queries in System R, *ACM Trans. on Database Systems*, 6(1):70-94 (March 1981).

```

$[                /* define record to read */
record {
    char *name;
    int age;
} foo;
]$

...
foo foorec;      /* declare a foo type record */
int curno;

...
$[    /* prepare to use foofile */
    attach { foofile use(read-write) } ;
    /* start retrieving it */
    start cursor (2) foofile ;
    /* read a record */
    fetch (2) count(1) ;
    end cursor { (2) } ;

]$

...
$[ response warn;      /* process the response */
    fetch cursor(curno)
    record(foo) to(foorec):
        printf("name is %s, age is %d\n",
            foorec.name, foorec.age);

]$

```

**Figure 2.** Sample module and response

Third, commands are sent to the iDBP to make it send back some data. In this example, one record is read from the file *foofile*.

Finally, a response block retrieves the data and processes it. In a real program there would probably be a loop to retrieve all of the records from the file as well as error detection and recovery code in the response block. Several cursors are typically active at once, so for each record returned the program would examine the cursor number, invoke the appropriate read routine and process the records returned.

#### 4. HIGHER LEVEL USER INTERFACE

Although the embedded language and *ddml* are adequate tools for the programmer and data-base analyst, they are much too low-level for the casual user. For this reason, we are working on several higher level interfaces that are more "user friendly."

##### 4.1 Screen oriented inquiry language

*Scrt*, the screen oriented inquiry language is intended to resemble the IBM Query-by-Example<sup>3</sup> language. It starts by drawing a blank tableau on the screen, into which the user types the name of the desired file. *Scrt* retrieves structure information from the iDBP about the file and fills in the column names in the tableau to correspond to the item names in the file. The user can then enter selection criteria, e.g. if only people whose ages are less than 50 are desired, he just enters 50 into the *age* column. Then *scrt* transforms the selection criteria into iDBP view creation commands and retrieves the contents of the view so created onto the screen.

3. Moshe M. Zloof. Query-by-Example: a data base language, *IBM Systems Journal*, 16(4):324-43 (1977).

---

We anticipate extending *scrt* to handle multi-file queries, query storage, and other such facilities. In our experience the QBE facilities for data entry and update are so unpleasant to use the nobody ever uses them; thus we do not expect to add such features to *scrt*.

#### **4.2 Litigation support system**

In cooperation with a large New York law firm, we have designed a litigation support system to track documents of interest in court cases. It resembles *Lexis* in that the full text of each document can be stored and selections can be made based on the contents of documents as well as dates, authors' names, and other criteria.

The litigation support system is in fact a general document retrieval system and would be of interest to any organization that needs to keep track of large numbers of documents.

The litigation support application is uniquely well supported by the iDBP. The iDBP's connected views and unstructured files give it the ability to deal conveniently and efficiently with large chunks of text, unlike other host-based and back-end data-base systems of which we know.

#### **4.3 Uniform user interface**

INTERACTIVE has for several years been developing a uniform screen-oriented interface to the IS/3 system.<sup>4</sup> We are evaluating approaches to integrating the iDBP with this package. Some work will be required to resolve incompatibilities between the current typeless scheme in the Uniform User Interface and the strongly typed data of the iDBP.

We expect that the power and flexibility of the Uniform User Interface will make it the method of choice for dealing with the iDBP, as it already is for other host-based applications.

### **5. DISCLAIMER**

The work described in this paper is, at this time, an advanced development project only. No representation is made as to whether this work will be available as a product and, if so, what the terms and conditions of such availability will be.

---

4. J. Spencer Rugaber. *A Uniform User Interface to UNIX*, presented at UNICOM, 1/26-28/83, San Diego, CA.

---

# **/rdb: A Relational Data Base Management System for UNIX**

*Rod Manis*

75 Buena Vista East, Suite 305  
San Francisco, CA 94117

With all the database packages available for UNIX these days, why write yet another? The speaker gave three reasons:

1. He wanted one he could afford; if this meant writing it himself, this implied that it should be easy to write.
2. It should be VERY easy to use (and able to be ported, say, to a personal computer).
3. It should nonetheless be very powerful.

The /rdb database system permits a user to create tables (including column headings) with a standard text editor. Columns are separated by tabs, and the first two lines of the table consist of, respectively, the column names and fields of dashes which define field widths.

All commands which query the database are simple programs invoked from the shell. This makes it easy to combine them with other UNIX utilities (e.g. *awk*, which can be used to perform selects).

---

/rdb: A relational Data Base Management System for Unix  
by Rod Manis, Computer Consulting  
1300-A space Park Way, Mountain View, CA 94043 (415)967-7770

/rdb\* is a relational data base management system for UNIX\*\*.  
/rdb is inexpensive, simple and powerful.

The price is only \$250. That is an order of magnitude less than the cheapest of the half dozen other relational data base systems for UNIX announced by January 83. /rdb is aimed at the coming mass market of microcomputer UNIX systems.

/rdb consists of over thirty programs that manipulate simple tables. The user can create a table in a text editor or from other tables. For example, a table named expenses might look like this:

| Date   | Amount | Account | Description              |
|--------|--------|---------|--------------------------|
| -----  | -----  | -----   | -----                    |
| 830125 | 67.00  | 4120    | UNICOM plane ticket      |
| 830126 | 12.00  | 4120    | UNICOM meal with client  |
| 830126 | .43    | 4121    | UNICOM bribe of official |
| 830127 | 150.00 | 4120    | UNICOM hotel bill        |

There are only three rules for the tables: 1. A tab is inserted between each column. 2. On the first line of the table each column is given a name. 3. The second line of the table has dash lines for each column showing the width of the column. this is much easier for the user than systems that require schema and special files.

Once tables are created, other tables can be produced from them by commands such as project (columns), select (rows by logical conditions), jointable (joins two tables together on a common key column), compute (calculates columns as a function of other columns), sortable (like UNIX sort, but knows about /rdb tables), total and subtotal, mailing labels, for letters, tax table, etc. /rdb also has a list format for data that is hard to fit into a table.

/rdb is relationally complete and has the uniform relational property according to the definition set forth by E.F. Codd in his 1981 Turing Award speech printed in Communications of the ACM, 25 No. 2 (February 1982). Also see C.J. Date, "An Introduction to Database Systems, Third Edition", 1981 and Volume II 1982.

\* /rdb is a trademark of Rod Manis, Computer Consulting  
\*\* UNIX is a trademark of Bell Laboratories

---

All commands operate at the UNIX shell level and all tables are files accessible and manipulatable by the UNIX utilities including the text editor. Therefore, the user does not lose the power of UNIX by "going into" the data base management system as with most other systems. /rdb was easy to implement and is easy to maintain because it uses UNIX utilities, especially awk.

In addition, /rdb has a set of C routines for manipulating tables which can also be purchased. /rdb can be used as a tool kit to develop other applications. By October 82, an accounting package and a MRP package had been built on /rdb.

By October 83 /rdb was running on the Cosmos super microcomputer (Unisoft V7 UNIX with Berkeley Enhancements, Motorola 68000, Multibus). It will be ported to as many other computers running UNIX as possible.



---

NAME

|             |                                                        |
|-------------|--------------------------------------------------------|
| col         | -one line descriptions of col routines                 |
| column      | -outputs selected columns (same as project)            |
| compress    | -remove leading and trailing spaces (reverse of justif |
| compute     | -calculates columns of a table from other column       |
| computedate | -adds days to current date and displays new date       |
| difference  | -outputs table of rows in table1 that are not in table |
| enter       | -append lines to a list without an editor              |
| entertable  | -append lines to a table without an editor             |
| howmany     | -displays the number of commands                       |
| intersect   | -outputs table of rows that are in both input tables   |
| jointable   | -join of two tables into one where keys match          |
| justify     | -justifies and fills the columns of a table            |
| label       | -print mailing labels from mail list                   |
| letter      | -print letters from form letter and mail list          |
| listtotable | -input file in list format and output in table format  |
| maximum     | -maximum value in a column                             |
| mean        | -average of a column                                   |
| menu        | -displays the commands available                       |
| minimum     | -minimum value in a column                             |
| number      | -add a column numbering each row of a table            |
| precision   | -maximum digits to right of decimal point in a column  |
| project     | -display columns of a table in any order               |
| rdb         | -describe available rdb commands (this list)           |
| rmcore      | -remove all core files to free up space on the disk    |
| rename      | -change name of column heads                           |
| row         | -outputs table with selected rows (same as select)     |
| select      | -output new table where rows match a condition         |
| sorttable   | -sorts a table by one or more columns                  |
| subtotal    | -outputs the subtotal of columns in a table            |
| tax         | -look up income in a tax table compute tax             |
| tabletolist | -input file in table format and output list format     |
| today'sdate | -output the current date in the format: 830615         |
| total       | -sums up each column selected and displays             |
| union       | -concatenate two tables with same column names         |
| validate    | -uses user created awk script to check input table     |
| what is     | -command description and syntax                        |
| width       | -width of the longest string in a column               |

---

## **Focus/USE: A Low Keystroke Database Editor and Browser**

*Anthony I. Wasserman*

Medical Information Science  
University of California, San Francisco  
San Francisco, CA 94143

*Martin Kersten*

Wiskundig Seminarium  
Vrije Universiteit  
de Boelelaan 1081  
Amsterdam, the Netherlands

It's often the case that, while query languages associated with database systems may be powerful enough to perform necessary operations, they are nonetheless not totally satisfactory because they are often overly verbose (i.e. require a lot of typing), and many people find them difficult to learn. Focus is a simple, low keystroke interface to the TROLL database system (both of which are outgrowths of the USE project [see the summary of RAPID elsewhere in these notes]). It was intended to facilitate both the addition and updating of records in a relational database system, and also browsing through the database and retrieval of records.

One of Focus's more interesting attributes is its ability to permit a user to 'focus in' on a desired set of records. One can select a set of tuples based on a specified condition, then select a subset of that set based on another condition, and so on, until one has gotten the set of records one wants. It is also possible to 'pop up' to previous sets which were selected during the focusing process (and which are therefore supersets of the current subset).

---

## Focus/USE: a Low Keystroke Database Editor and Browser

Anthony I. Wasserman

Medical Information Science  
University of California, San Francisco  
San Francisco, CA 94143

Martin Kersten

Wiskundig Seminarium  
Vrije Universiteit  
de Boelelaan 1081  
Amsterdam, the Netherlands

### 1. Introduction

Most database systems provide an interactive query language for selective retrieval and display of data, and for modifications (including new insertions) to the database. However, such query languages tend to be difficult to learn and remember, as well as requiring a significant amount of typing.

Instead, we have created a simple, low keystroke, screen-oriented interface to our Troll relational DBMS [1,2], to complement its other interfaces. Our goal was to make this tool suitable both for relation editing (updates and insertions on a tuple level) and for browsing/retrieval.

We observed that many retrieval sequences involve increasing selectivity on tuple sets. Users select a set of tuples based on a condition, examine that set, select a subset of that set, and so on, until the desired set is found. The user then wants to display or save that set. This "focusing" approach to database browsing seemed so natural and commonplace that we made it a central feature of our tool.

This tool, named Focus/USE, has been designed to work in conjunction with the Troll relational DBMS as one of the User Software Engineering [3] tools, and has been implemented as a front-end for Troll. Space limitations make it impossible to present all of the features of Focus/USE, which are described more fully in [4]. We concentrate here on the "focusing" features.

### 2. Browsing and Retrieval Capabilities of Focus/USE

We describe the capabilities of Focus/USE by a series of examples using a relation of LP records, defined as follows:

```
relation LPs [key Title, Artist, Label] of
  Title, Artist, Label, Albumnumber: string;
  Year: integer (1949..1999);
  Category: scalar (classical, jazz, country, folk, blues, soul, rock,
                    show, pop, gospel, comedy, other);
end;
```

Focus/USE works with one relation at a time. When that relation is given, the (first 16) attributes of that relation are displayed vertically on the screen (one attribute per line). Retrieval and browsing commands fill in values next to each attribute for each tuple. If a relation has more than 16 attributes, the page forward ("F") and page back ("B") commands provide access to the entire tuple.

The **get** command can be used to obtain a single tuple, and the **focus** command can be used to obtain a tuple set, called a partition. Successive focus commands can

---

be given on a partition to yield continual refinement on the tuple set.

For the **get** command, values are supplied for all key attributes in a relation. If the tuple is found with those key attribute values, Focus fills in the other known attributes of the tuple from the relation. The displayed tuple can then be edited if desired.

More often, though, users wish to retrieve the tuples that meet a stated constraint, then browse through the selected set. The **n** (next) and **p** (previous) commands permit sequential examination of tuples, and the **A** (aggregate) command provides summary data for attributes of the selected partition. Whenever a tuple is displayed, it may be edited and written out. (A "read only" option may be set, though.)

The **focus** command is the key to these operations. For string-valued attributes, a substring may be provided to retrieve all tuples on which the partial match is successful. The comparison operators (**=**, **~**, **>**, **<**, **>=**, **<=**) may be used on numeric (integer and float), string, and scalar attributes are accessed with a value to retrieve all tuples on which the comparison is successful.

In this way, a set of values or partial values can be inserted on a clear screen, and a focus command used to retrieve the desired partition. Thus, the input "Who" in the attribute field for Artist in the LP relation would create a partition containing all tuples where the Artist attribute contained the string "Who". (The Who and Guess Who would qualify, for example.)

As another example, one could select all LPs from before 1968 by moving the cursor to the Year attribute, and typing "< 1968" on the line.

## 2.1. Multiple Levels of Focusing

Compound conditions may be combined into a single focus or divided into two or more steps. Thus, one could focus on all of the LPs where Artist contains the string 'Stones' with Year < 1969 by inserting the appropriate information in the Artist and Year attributes, then giving the **focus** command.

Alternatively, one could focus on one condition and *then make a second focus on the selected partition*. The focus mechanism may be used up to 10 levels of depth, making successive refinements to the selected set until no tuples are selected. It is always possible to return to the previous focus level by typing "q", and to exit the entire set of focuses by typing "Q". Furthermore, it is always possible to display the focus condition that was given at each level.

## 2.2. Conjunction, disjunction, and negation in focuses

In addition to conjunction ('and') of conditions, Focus also allows disjunction ('or') and negation ('not'). Negation may be applied to any focus simply by typing the not sign ('~') on a clear screen. It will create a new focus at level K+1 containing all those tuples of the underlying relation that are not members of the partition of focus K.

Alternation of conditions may be done in several ways. First, if one wishes to select tuples satisfying at least one condition out of one or more placed on one or more attributes, the **|** command can be used. Thus, to select all LPs by the Beatles *and* all LPs with a date prior to 1966 (inclusive or), type "Beatles" in the Artist field, "<1966" in the Year field, then type **|** to obtain the desired partition.

The "set" specification may be used to do alternation on a single attribute. Instead of first focusing on Year >= 1964 and then focusing further on Year <= 1969, the retrieval can be done with only one focus operation by typing

---

[1964,1965,1966,1967,1968,1969]

[1964..1969]

in the attribute field, then doing a focus.

The set specification also causes a partial match on strings. Thus, if one typed

[Four, Five]

in the attribute field for Artist and did a focus, the partition would contain all tuples of LP where Artist contains the string 'Four' or 'Five', e.g., Four Tops, Brothers Four, Five Satins, We Five, Frankie Valli and the Four Seasons.

### 2.3. Saving Focus Results

Focus partitions and their conditions may be saved for further processing by using the **copy**, **list**, and **screen** commands. In these formats, tuples can be used for a variety of display and storage functions. Thus, Focus/USE provides a quick and convenient method for carrying out common database retrievals and passing the results to other programs.

### 3. Conclusion

Focus/USE, along with Troll, is part of the User Software Engineering distribution made by the University of California, San Francisco, and the Vrije Universiteit, Amsterdam. The tools presently work on VAX Unix systems and on PDP-11's having separate I-and-D space.

Further work is underway on Focus/USE to provide a view capability, and to make both Troll and Focus/USE available on other computer systems running Unix, particularly those based on the Motorola 68000.

### 4. References

- [1] M.L. Kersten and A.I. Wasserman, "The Architecture of the PLAIN Data Base Handler," *Software -- Practice and Experience*, vol. 11, no. 2 (February, 1981), pp. 175-186.
- [2] A.I. Wasserman and M.L. Kersten, "A Troll Tutorial", Laboratory of Medical Information Science, University of California San Francisco, and Wiskundig Seminarium, Vrije Univeristeit, Amsterdam, 1982.
- [3] A.I. Wasserman, "The User Software Engineering Methodology: an Overview," in *Information System Design Methodologies: a Comparative Review*, ed. T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart. Amsterdam: North Holland, 1982, pp. 589-628.
- [4] A.I. Wasserman, M.L. Kersten, and C. Resnikoff, "A Focus/USE Tutorial", Laboratory of Medical Information Science, University of California San Francisco, and Wiskundig Seminarium, Vrije Univeristeit, Amsterdam, 1982.

---

# The Informix Commercial DBMS for UNIX

*Laura L. King*

Relational Database Systems, Inc.  
1208 Apollo Way, Suite 503  
Sunnyvale, CA 94086

In this paper the *informix* relational database management system is described as the ideal tool for the building of computer applications which are designed to gather business, engineering or scientific data and produce informative reports. Use of *informix* by end-users, system integrators and application builders is discussed.

The *c-isam* B-tree based retrieval method is described as being *informix*'s software search and sort engine, whose features are critical for end-user ease and flexibility. *C-isam* is also described as the up and coming *de facto* ISAM for UNIX operating systems, and the C language.

The *ace* relational report writing language is described as an English-like and non-procedural tool for producing 100% of the needed reports from a database, in exactly the format the user desires. The fact that *ace* is a non-procedural language is cited as the reason for the greater than ten-to-one gain in man-hours needed for report development and maintenance.

The new *perform* custom screen building system is now being shipped to *informix* users. The user or system builder can construct *perform* screens for data entry and maintenance, as well as "query by example" style database interrogation.

---

**This page intentionally left blank**

Chairperson: *Phil Cohen*  
Institution Missing

## Device Independent Graphics Enhancements at ITTDCD

*Greg Hidley*

ITT Defense Communications Division  
10060 Carroll Canyon Rd.  
San Diego, CA 92131

As of the 4.1 Berkeley Software Distribution the graphics packages available under UNIX had certain limitations: *vplot*, the Versatec interface, did not run on the 32 bit machine and did not share the Versatec as did other support software (*vpr*, *vtroff*, *vgrind*, etc.). The UNIX plot library (*-lplot*) did not do any form of automatic scaling and did not allow optimized scaling for plots which were overly X or Y range dominated. Graph offers additional flexibility as a front end for standard two dimensional Cartesian plots but does not allow user defined subroutines for the generation of non-standard representations often needed in a research environment.

In addition to modifying *vplot* to work on the 32 bit machine, queuing plots for the Versatec, and adding "plot handlers" for the HP Penplotter, Matrox graphics monitor, HP 2648a, Dec VS11, and Digital Engineering VT100 Retro graphics terminals, we modified the plot library to allow automatic scaling, user defined scaling, and implemented the functions *circle* and *arc*. There is also a plot handler for a non-graphics terminal which uses *curses* and *termcap* for a rough approximation of the plot.

We enhanced the UNIX *plot* command's inherent device independency by using user-assigned environment variables to control the run time output of plotting programs. Thus, a user could write a single C program to plot a graph and, when loaded with the ITT plot library, could execute that single program and get plots on any of the devices listed above. The environment variable PLOTTER would determine at runtime the destination of the plot. If hardcopy devices were targeted automatic queueing would be invoked.

The package is compatible with the standard UNIX plot library routines and the plot handlers are usable with the standard UNIX intermediate plotfiles. Scaling is handled by scanning the coordinate ranges while building the intermediate plotfile and then backpatching appropriate deltas at the beginning of the file. The plot handlers use independent scaling macros for the *x* and *y* axis which allow both scaling that either retains shape or that optimizes the plotting surface of the target device. The former scaling is default but can be overridden by the user.

We are currently implementing an interactive query system which would allow the user to view a sequence of plots on a graphics terminal and redirect copies of selected plots to a hardcopy output.



---

## Device Independent Graphics Enhancements at ITTDCD

*Gregory Hidley*

ITT DEFENSE COMMUNICATIONS  
10060 Carroll Canyon Rd.  
San Diego, Calif. 92131

### 1. Introduction

The Unix\* 4.1 Berkeley Software Distribution graphics packages reflect a solid design which supports ease of usage, target device independency, and extensibility of features. This package, however, has certain limitations: vplot, the Versatec interface, does not run on the 32 bit machine and does not share the Versatec (via queueing) as does other support software (vpr, vtroff, vgrind etc.). The Unix plot library (-lplot) does not do any form of automatic scaling and does not allow optimized scaling for plots which are overly X or Y range dominated. Although the "graph" front end partially overcomes these limitations for standard two dimensional Cartesian plots, it still does not allow user defined subroutines required for the generation of those non-standard representations often needed in a research environment. {slide 1}

The enhancements made at ITT facilitate 1) ease of usage; 2) downward compatibility with existing Unix plot software and library routines; 3) target device independence; 4) optimization of system resources; and 5) ease of expansion to include new graphics devices. To make this presentation clearer I will first review the standard Unix plotting mechanism and then explain in more detail the design and implementation of these enhancements. {slide 2}

### 2. Standard Unix Plot Generation

Standard Unix plot generation consists of two tasks: 1) the generation of an intermediate device independent representation of the plot; and 2) the interpretation and plotting of that intermediate representation by an appropriate plot device "handler" for the target device {slide 3}. Step 1 is accomplished by either the usage of a front end program such as "graph" or a user program which takes advantage of Unix plot library primitives such as "line()", "point()", "circle()" etc. The second step uses the standard "plot" program as a mechanism for the selection and invocation of the appropriate device handler. The user's role in device selection consists of choosing the "plot" flag for the appropriate device: "-T450" for a Diablo type printer; "-T4015" for a Tektroniks graphics terminal; or "-Tver" for the Versatec printer/plotter.

### 3. ITTDCD Enhancements

The additional flags "-Tmtx", "-Tvs11", "-Thp", "-Tretro", "-Tpen", "-Tcrt", and "-Tdebug" are available for plotting on the Matrox, DEC VS11, HP 2648, and Digital Engineering Retro graphics terminals, the HP penplotter, and a standard non-graphics CRT (with the help of "curses" and "termcap") respectively. "-Tdebug" gives the user diagnostic information useful for debugging a program's graphic output.

---

\*Unix is a trademark of Bell Laboratories

---

#### **4. Ease of Usage and Downward Compatibility**

From the user's point of view the standard Unix plotting documentation still applies. The package is compatible with the standard Unix plot library routines and the plot handlers are usable with the standard Unix intermediate plot files. All of the original "plot" program device flags are still present. {slide 4}

The user who wishes to write his own plotting programs can do so with the extended plot library routines which include such additional features as selection of pen colors (for appropriate devices), user defined scaling, automatic scaling, and arc and circle routines.

#### **5. Target Device Independence**

We enhanced the Unix "plot" command's inherent device independence by using user assigned environment variables to control the run time output of plotting programs. Thus, a user could write a single C program to plot a graph and, when loaded with the enhanced plot library, could execute that program to produce plots on any of the above mentioned devices with neither recompilation nor relinking. The environment variable PLOTTER would determine at runtime the destination of the plot. This mechanism works also for front ends which use the "plot" command directly. The absence of the "-T" flag is a signal to extract the target device from the user's environment variable as above. If none is set the default value is determined by checking the "TERM" environment variable and choosing the appropriate handler for the terminal in use. When hard copy devices are targetted automatic queueing takes place.

#### **6. Optimization of System Resources**

Two disadvantages of the distributed "vplot" Versatec handler (in addition to not having been modified to run on the 32 bit Vax in the first place) are its inability to share the Versatec through queueing output (as do "vpr", and "vtroff"), and its lack of an effective locking mechanism for the use of its raster file (as distributed "vplot" used the fixed file /usr/tmp/raster).

In our plotting package "vplot", as well as other programs which build raster files for the Versatec, use a double queueing mechanism: one to build the raster file; the other to print it on the appropriate Versatec. The first queue determines how much free space is available on the /usr file system where the raster spooling area resides and allows the building of as many raster files as can be accommodated before reaching a free space lower threshold. Once the raster file has been built it is linked into the "vpr" queue just as any other "vtroff" and "vpr" program. The existing Versatec support software "vpq" and "vprm" have been modified to give detailed queue status on both queues and allow the removal of the job from either queue. We have also modified "vpd" to print the plots in time order rather than random directory slot order. These capabilities give the user control over the status and progress of the plot, allow the simultaneous generation of as many raster files as space allows, and optimize the usage of the Versatec.

#### **7. Ease of Expansion**

A device handler template based on the structure of vplot has been used to ease the addition of new plotting devices. The device dependent portion of the code has been minimized to two routines which plot points and lines. Once a plot handler is written for the new device, a one line change is all that is needed to modify the "plot" shell script and plot library to link in that new device.

---

## 8. Design

The current package furthers the Unix philosophy of device independent programming. What we have done is to take the basic designs of the "plot" program and "vpr" software and expanded them to work with additional devices. The same can be said of the Unix plot library. The use of environment variables to control the output fits easily into the program and affords much freedom to the plot user when viewing or programming plots.

Scaling is handled by scanning the coordinate ranges while building the intermediate plot file and then backpatching appropriate deltas at the beginning of the file. This allows a two pass job to be done in one pass. The plot handlers use independent scaling macros for the X and Y axis which allows scaling for either the retention of shape or the optimization of the plotting surface. The former scaling is the default option but can be overridden by the user. {slide 5}

An additional advantage to our package is the consistency of the conventions used for queueing software. The basic design again is patterned after the existing "vpr" software. Just as the "vpr" package made use of the support routines "vpf", "vpd", "vpq" and "vprm" to build, monitor, and maintain output destined for the Versatec, so do our "ppr" (penplotter queue) and "rfr" (raster file queue) packages use the same variants (i.e. "pfr", "pfd", "ppq", "pprm", "rfr", and "rfd").

## 9. Planned Enhancements

We are currently implementing an interactive previewing capability which would allow the user to view a sequence of plots on a graphics terminal and redirect copies of selected plots to a hardcopy output device. A user defined sub-window mechanism is planned which will allow the user to plot multiple plots on one page. These could be adjacent or overlapping.

---

# **Terminal-Independent Plotting Packages**

## **An Example and Suggestions for Standards**

*Don Mackay*

University of California at San Diego  
Department of Chemistry B-014  
La Jolla, CA 92093

Crude plotting on character terminals is a quick and convenient way to display data when a high resolution device is unavailable or inappropriate. As enhanced graphics capabilities make their way into low priced terminals, it becomes desirable to take advantage of line drawing and other special characters for low resolution plots on terminals. A sample set of terminal independent plotting programs currently in use at UCSD is discussed in terms of the variety of enhancements used and the method of implementation. Suggestions for standardization of graphics attributes in the *termcap* or *terminfo* files are presented.

---

## Terminal\* Independent Plotting Packages: An Example and Suggestions for Standards

*Don Mackay*

Department of Chemistry  
University of California, San Diego  
La Jolla, California 92093

### Introduction

Low resolution plotting on character terminals is a quick, convenient, and general way to display graphical data when a high resolution device is unavailable, inconvenient, or inappropriate. As enhanced character graphics make their way into low priced terminals (in the way of line drawing and more recently, "block" graphics), it becomes desirable to take advantage of these special characters for low resolution plots on terminals. Preferring the philosophy that software should extract the maximum performance from hardware rather than settle for the mundane "least common denominator", but also recognizing the necessity for generality, we in the chemistry department at UCSD have implemented a terminal independent plotting package (TPLOT) which allows any character printing device to utilize all its graphics capabilities while performing as a low resolution plotter. As a result, we have found that our need for high resolution graphics and hard copy plots have been drastically reduced as we can do most previewing, debugging, and spot checking with rapid convenient enhanced character representations.

Because of the large diversity of terminal models in a typical UNIX† environment (ours is no exception), terminal independence is a very desirable characteristic in a terminal plotting package. The principle mechanism in Berkeley UNIX for terminal independent programs is the *termcap* file and its associated utilities (see *TERMCAP*(3,5) in the *UNIX Programmer's Manual*). As *TERMCAP*(5) describes, the specific control codes and specifications of every terminal are associated with the general *termcap* entries for those capabilities and stored in a database (*/etc/termcap*) for that terminal. Subroutines are available (*TERMCAP*(3)) which can determine precisely what kind of terminal is currently in use and inform a program if indeed the terminal has a given capability and if present, what the appropriate control code or specification is. While *TERMCAP*(5) describes an extensive assembly of the common character terminal capabilities, missing are any entries to allow for graphics capabilities. Fortunately, it is very simple to add new entries to the *termcap* database and we have in fact done so. Hopefully, our implementation can serve as a model that others can follow (or better yet, that can be adopted by Berkeley) so that terminal independent low resolution plotting programs can be more portable.

### Implementation

There are three basic groups of enhanced capabilities one would want to employ in a basic terminal plotting program: *i)* video attributes, *ii)* line drawing characters, and *iii)* block graphics. Table 1 lists in a *TERMCAP*(5) format the enhanced graphics *termcap* entries that we have provided to describe the most commonly available graphics characters and control codes.

---

\*Terminal here refers to any character generating device but only to bit mapped or stroke devices when operating in a "character" mode.

†UNIX is a Trademark of Bell Laboratories.

**Video attributes.** Video attributes such as reverse video, half intensity, blinking, etc, can greatly enhance the readability of graphs. Video attributes are already available through the standard termcap distribution and so no detailed description of them will be made here. Video enhancements can be used to distinguish different lines on the same graph. This is especially useful when creating lines from block graphics where the multiple lines cannot be distinguished simply by the type of character from which they are constructed. This can be implemented by defining alternate "enter/exit" sequences which have video attributes embedded in them. For this reason, we have included in table 1 several ways to enter graphics mode.

**Table 1. TERMCAP Database Graphics Extensions**

| <i>Name</i> | <i>Type</i> | <i>Description</i>                                    |
|-------------|-------------|-------------------------------------------------------|
| gr          | str         | enter graphics mode                                   |
| gx          | str         | enter graphics mode (with alternate video attributes) |
| gy          | str         | enter graphics mode (with alternate video attributes) |
| gz          | str         | enter graphics mode (with alternate video attributes) |
| ge          | str         | exit graphics mode                                    |
| gp          | str         | graphics mode padding sequence                        |
| gA          | str         | horizontal line segment                               |
| gB          | str         | vertical line segment                                 |
| gC          | str         | top left-hand corner                                  |
| gD          | str         | top right-hand corner                                 |
| gE          | str         | bottom left-hand corner                               |
| gF          | str         | bottom right-hand corner                              |
| gG          | str         | crossed lines                                         |
| gH          | str         | left intersect                                        |
| gI          | str         | right intersect                                       |
| gJ          | str         | bottom intersect                                      |
| gK          | str         | top intersect                                         |
| g0          | num         | mask value for block graphics                         |
| g1          | num         | top left sub-cell                                     |
| g2          | num         | top right sub-cell                                    |
| g3          | num         | middle left sub-cell                                  |
| g4          | num         | middle right sub-cell                                 |
| g5          | num         | bottom left sub-cell                                  |
| g6          | num         | bottom right sub-cell                                 |

**Line drawing characters.** Many terminals have an alternate character set which provides line segments (vertical, horizontal, intersections, etc.) with which to build aesthetically appealing rectangular shapes. While they are of limited usefulness in improving resolution, they do improve overall appearance and readability. The specific names and assignments we have used is given in the center section of table 1.

There are two commonly used approaches to enabling such alternate character sets. Either the new character set can be enabled/disabled with enter/exit control codes, thereby mapping all intervening output into the new character set, or each character is padded with a control code turning on the graphics character set for only that character. Thus, provisions have been made for the first case with "enter/exit" sequences (gr,gx,gy,gz/ge) and for the second with a "padding" sequence (gp).

**Block graphics characters.** Recently, several terminal manufactures have offered "block" or "business" graphics. Essentially, the character cell is divided into a 2 by 3 grid of smaller sub-cells and an alternate character set containing all 65 combinations of these sub-cells is enabled for drawing psuedo medium resolution. This scheme increases the effective resolution of a standard 24 line by 80 column terminal to 72x160. Fortunately, the assignment of the various possible configurations is done systematically so that "turning on" one of the sub-cells is equivalent to "turning on" a specific bit in the character byte (see figure 1). Depending on the manufacturer, an offset or mask operation may be required as well (g0 in table 1). The block graphics character set typically contains the line drawing characters also and therefore

**Fig. 1.** The subdivision of the character cell for block graphics. The codes g1-6 correspond to the termcap names for the various sub-cells (see table 1). The numbers in parentheses are the appropriate (octal) values for a FALCO TS-1E terminal.

|             |             |
|-------------|-------------|
| g1<br>(040) | g2<br>(020) |
| g3<br>(010) | g4<br>(004) |
| g5<br>(002) | g6<br>(001) |

does not require a different enter/exit control code. For the sake of portability however, if different character sets are involved, the "gr" sequence should be reserved for line drawing graphics.

In essence then, simple vector to raster algorithms can be implemented by simply "turning on" (using simple boolean logic) the appropriate bits of a character buffer for those sub-cells that the vector crosses. In practice, it is convenient to perform all the desired mappings on a "screen buffer" (*i.e.*, a matrix of 24 by 80 characters representing the terminal screen) since after a character cell has been output to the terminal, enabling another sub-cell in that same character location will require knowing what sub-cells are already enabled to avoid erasing existing data. Table 1 lists the termcap entries we have identified as being sufficient to handle the case of block graphics. Note that we have taken advantage of the bit wise organization of the sub-cell assignments so that 6 entries are sufficient to define all 65 possible block graphics characters. The extra entry (g0) is provided for a mask value.

### Examples

Below are examples directly from our termcap file of the entries for two representative terminals, an AMPEX Dialogue 80 which can do line drawing but not block graphics, and a FALCO TS1-E which can do both (the pertinent graphics entries are in bold). It should be noted that our terminal independent plotting package will still function in the absence of any graphics entries in the termcap file by using as a default the standard ASCII character set and in fact, if output is directed to a file (so that it is impossible to know what terminal may eventually display the file), the default format is used.

Sample TERMCAP Entries (graphics in bold)

```
TA&mpex;d80;dialogue80&mpex dialogue 80:\
:am:ca:bs:pt:cl=\E*:cm=\E=%+ %+ :rv=\Ej:re=\Ek:al=\EE:ho=\O36:\
:bt=\El:ic=\EQ:im=:ei=:dl=\ER:dc=\EW:ha=\E):he=\E(:ce=\Et:\
:cd=\Ey:so=\Ej\En:se=\Ek\Eo:li#24:co#80:nd=^L:up=^K:us=\El:\
:ue=\Em:kl=^h:kr=^l:ku=^k:kd=^j:kh=\O36:ma=^hh^ll^kk^jj:\
:gr=\200:ge=\200:gp=\EG:gA=I:gB=J:gC=A:gD=B:gE=C:gF=D:\
:gG=K:gH=G:gI=F:gJ=H:gK=E:
fa[ts]falc[falco ts-1e:ca:\
:al=\EE:am:bs:ce=\EF1\EK\ET:cl=\E*:cm=\E=%+ %+ :co#80:\
:dc=\EW:dl=\ER:ho=\E[H:im=\Eq:ei=\Er:cs=\Es%+ %+ :mi:\
:kl=\E[D:kr=\E[C:ku=\E[A:kd=\E[B:kh=\E[H:li#24:nd=\E[C:\
:pt:bt=\El:up=\E[A:us=\Eg1:ue=\Ef:so=\E(:se=\E):\
:is=\E(042\E3\E)\ED0\Eg0\E.^n^a\E.^o^p:xs:rv=\Eg4:re=\Eg0:\
:gr=\Eg8:gx=\Eg8\E):gy=\EgA\E(:gz=\EgA:ge=\Eg0\E):gp=\200:\
:gA=\E\22:gB=\E\31:gC=\E\15:gD=\E\14:gE=\E\16:\
:gF=\E\13:gG=\E\17:gH=\E\25:gI=\E\26:gJ=\E\27:gK=\E\30:\
:g0#64:g1#32:g2#16:g3#8:g4#4:g5#2:g6#1:ha=\E):he=\E(:
```

Using the various character graphics features in plotting programs significantly enhances both the readability and resolution of data graphs on terminals. Figure 2 shows actual graphs on various models of terminals using our terminal independent graphics package (TPLOT). As can be seen, the block graphics is a significant improvement over the simple character style plot although even the character resolution plots still convey enough information to be useful. The block graphics provide a reasonable compromise between the ASCII character plots and bit

mapped graphics.

TPLOT is a set of high level subroutines which provide single calls for drawing common graph structures: axes (with tics and labels), grids, lines (connecting user supplied data points), and strings. They are compatible on a call by call basis with an existing set of vector graphics routines used in our facility (GPlot). This allows a user program to switch between a vector and a character device format by simply linking with the appropriate library. Largely because of its extensive use of the termcap facilities, TPLOT will print on any terminal using video enhancements, line drawing, and block graphics if available. It takes advantage of an addressable cursor, wider and longer screens, and its output can be saved in files or redirected to other printing devices (such as line printers). While the TPLOT routines represent a rather specific application of terminal independent graphics, the incorporation of graphics character entries into the termcap file has proven to be extremely useful and should be equally valuable in other applications.

### Conclusion

Until we all have a bit mapped terminal at our desks (which we all hope is soon), there will be a continued need to utilize existing terminal resources to their fullest potential. Character resolution plotting provides a convenient and often sufficient way to display data and therefore is a reasonable and practical use of inexpensive terminals. By fully utilizing the termcap utilities (including the graphics enhancements outlined in this article), terminal *dependent* software can be replaced by plotting software that is general, powerful, and portable.

### Acknowledgements

I would like to acknowledge Peter H. Berens, whose criticism and helpful suggestions prompted me to implement these ideas.

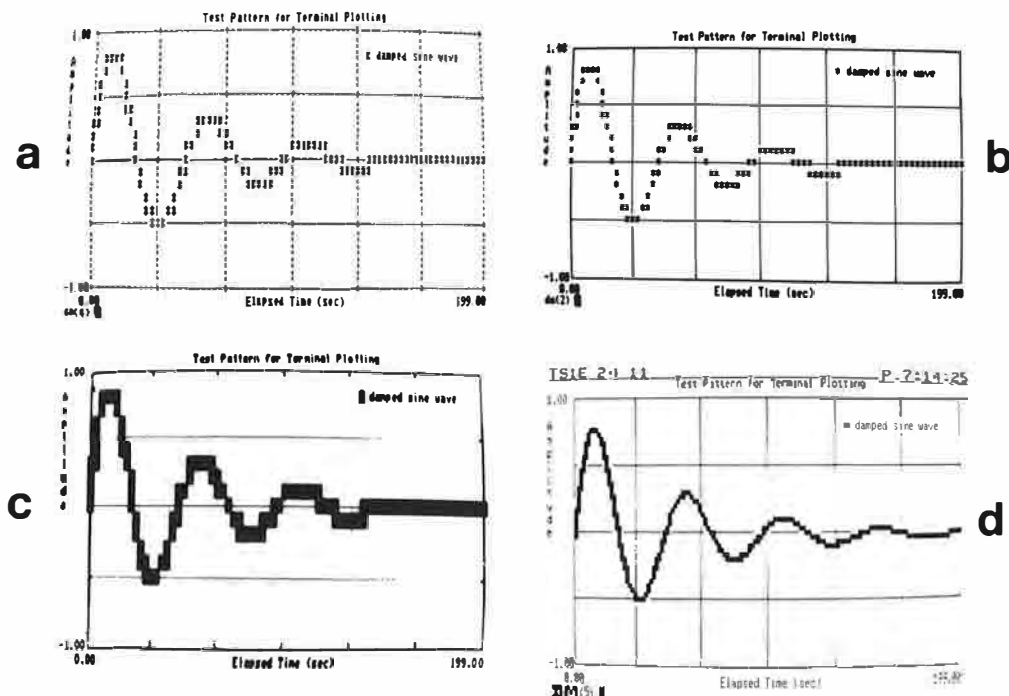


Fig. 2. Sample outputs of the TPLOT terminal independent plotting package created with identical input data on four different terminals: a) ADM-3A, b) Visual 200, c) Dialogue 80, and d) Falco TS-1E. Plot a represents an example with no graphics or video enhancements. Plot b shows the improvement resulting from including line drawing graphics and half intensity (grid lines). Plot c includes line drawing, half intensity, and reverse video (data curve). Plot d shows the improved resolution provided by block graphics. Note that the "tail" of graph d still contains structural detail where the previous representations have none.



---

**This page intentionally left blank**

---

# Graphics Standards for Personal Workstations

*Michael Shantz*

Sun Microsystems, Inc.  
2310 Walsh Avenue  
Santa Clara, CA 95051

*Graphics workstation architecture.* A modern workstation architecture with local area network support provides an enhanced environment for engineering graphics applications. The Sun workstation has a 1K by 1K bitmap display with rasterop hardware, a 640 by 480 color frame buffer, one or two Mbytes of main memory, a roughly 1 MIP MC68000 processor, an Ethernet interface, and vector drawing at 3 to 4 microseconds per point. The 4.2BSD UNIX operating system provides an advanced network interface with remote shell and remote login, 10 MHz network and a hardware graphics output pipeline will complete the low-cost high-performance graphics workstation environment.

*Core, GKS, and VDM.* Adherence to standards for both hardware and software is central to the design philosophy of the Sun workstation. Hardware "standards" for this workstation include the Multibus which facilitates system expansion, the Ethernet which facilitates multi-host local networks, the MC68000 processor, and SMD disk interfaces. Software "standards" used include UNIX, the C programming language, and for graphics, the ACM SIGGRAPH Core system, the GKS graphics standard, and the Virtual Device Metafile (VDM) draft proposed standard. Graphics standards are important for device independence, software portability to future products, and for communication of graphical data.

We have implemented the ACM SIGGRAPH Core graphics package and a GKS implementation is scheduled for completion in mid '83. Each of these will have a VDM standard interface pending finalization of the standard. SunCore and SunGKS comprise a set of low level graphics output primitives such as lines, text, and filled regions, facilities for grouping primitives into segments which may be manipulated as a unit, and a set of input primitives that use the mouse and keyboard for dragging, drawing and transforming segments. Device independence allows the user program to direct graphics output to the bitmap, color display, VDM, or to a process on another workstation on the ethernet. Hardcopy output requires piping VDM data over the network to a process on the printer server.

*Extensions to the standards.* Planned extensions to the graphics standards implementation include integrating the graphics processes and the window system so that graphics processes and other UNIX processes can execute simultaneously in multiple windows. Shaded surface polygon primitives with hidden surface elimination via a virtual memory z-buffer and bicubic patches with image and texture mapping are planned for '83. Other raster extensions will be added for handling bitmaps and imagery in a Core compatible manner. Interprocess communication and file transfers between workstations would permit sharing of large data bases and utilization of compute power or special hardware on other "file servers" or "crunch servers."

---

# Graphics Standards for Personal Workstations

Michael J. Shantz and Jerald R. Evans  
Sun Microsystems, Inc.  
2550 Garcia Ave., Mountain View, CA. 94043  
415-360-1300 ucbvax!ucbarpa!sun!shantz

## ABSTRACT

### Graphics workstation architecture

A modern workstation architecture with local area network support provides an enhanced environment for engineering graphics applications. The Sun Workstation has a 1024 by 800 bitmap display with rasterop hardware, a 640 by 480 color frame buffer, one to four Mbytes of main memory, a 10MHz MC68010 processor, an Ethernet interface, SMD disk controller, and Multibus. Vector drawing speed is 3 to 4 microseconds per pixel with much faster rates for horizontal or vertical vectors. The 4.2 bsd UNIX operating system provides an advanced network interface with remote shell, remote login, 10 Mbps network data rates, virtual memory, and improved disk I/O. Floating point hardware will complete the low-cost, high-performance graphics workstation environment.

### Core, GKS, and VDM

Adherence to standards for both hardware and software is central to the design philosophy of the Sun Workstation. Hardware 'standards' for this workstation include the Multibus which facilitates system expansion, the Ethernet which facilitates multi-host local networks, the MC68000 processor, and SMD disk interfaces. Software 'standards' used include UNIX, the C programming language and, for graphics, the ACM SIGGRAPH Core system, the GKS graphics standard, and the Virtual Device Metafile (VDM) draft proposed standard. Graphics standards are important for device independence, software portability to future products, and for communication of graphical data.

We have implemented the ACM SIGGRAPH Core graphics system and a GKS implementation is scheduled for completion in mid 1983. Each of these will have a VDM standard interface pending finalization of the standard. SunCore and SunGKS comprise a set of low level graphics output primitives such as lines, text, and filled regions, facilities for grouping primitives into segments which may be manipulated as a unit, and a set of input primitives which utilize the mouse and keyboard for dragging, drawing and transforming segments. Device independence allows the user program to direct graphics output to the bitmap, color display, VDM, or to a process on another workstation on the Ethernet. Hardcopy output requires piping VDM data over the network to a process on the printer server.

---

Sun Workstation and SunCore are trademarks of Sun Microsystems, Inc. Ethernet is a trademark of Xerox Corporation. Multibus is a trademark of Intel Corporation. UNIX is a trademark of Bell Laboratories.

---

The graphics processes and the window system are integrated so that graphics processes and other UNIX processes can execute simultaneously in multiple overlapping windows.

#### Extensions to the standards

Shaded surface 3-D polygon primitives have been added to SunCore and hidden surface elimination via a virtual memory z-buffer is planned for 1983. Other raster extensions are being added for handling bitmaps and imagery in a Core-compatible manner. Interprocess communication and file transfers between workstations permit sharing of large data bases and utilization of compute power or special hardware on other 'file servers' or 'crunch servers'. As computer graphics and image processing continue to merge, peripherals such as array processors, video digitizers, and pipeline image processors could be located at selected workstations on the net with access by processes on other workstations.

Large application design projects encounter the same problems of scale that large software development projects have, namely, how to control access to and modification of the pieces of a complex object which is being manipulated simultaneously by many engineers. Source code control systems, good and bad, have arisen to assist large software projects. Controlled check in and check out of the pieces of a large computer graphics data base is also desirable. The local area network Sun Workstation is aimed at providing the proper environment for handling such large graphics design projects.

---

# Windows with 4.2BSD

*Steven R. Evans*

Sun Microsystems, Incorporated  
2310 Walsh Avenue  
Santa Clara, CA 95051

The presentation describes the user interface and technical highlights of a UNIX-based multi-window development environment. The hardware environment of this system consists of the Sun workstation which utilizes a 10 MHz 68000 cpu, 1 MByte RAM, large bitmap display, optical mouse and rigid disk (possibly shared over a 10 MBit Ethernet). The software environment of this system is the new Berkeley 4.2BSD version of UNIX.

One of the most interesting problems facing a designer of what is intended to be a multi-window system for UNIX is how to achieve a reasonable level of cooperation between windows. This is troublesome because an application running in a window often knows nothing of the existence of other windows, to say nothing of the window system itself. Furthermore, a process can't communicate with the other processes in other windows due to separate address spaces unless a lot of Kernel support is utilized.

This talk describes what a "reasonable level of cooperation" is thought by the author to be. Also, the talk describes how the new facilities of 4.2BSD are utilized to implement the user interface ideas without requiring a great deal of operating system support.

---

## **Computer Animation at UCSD**

*Jeff Loomis*

Seacoast Software and Salk Institute

*Phil Mercurio*

Quantitative Morphology Lab  
University of California at San Diego

At UCSD's Quantitative Morphology Lab, UNIX is used for the control of high performance computer graphics hardware interfaced to a film animation bench. Our animation system has been used to produce scientific films in the areas of neurosciences and linguistics. Several artists have also been turned loose on the system with interesting results.

The presentation will include a brief description of the evolution of the hardware and software and how we took advantage of UNIX to save programming effort. We'll also be showing a short film made especially for the conference.

---

## COMPUTER ANIMATION AT UCSD

Jeffrey A. Loomis  
Alcyon Corporation  
8716 Production Ave.  
San Diego, CA 92121  
(619) 578-0860

ucbvax!sdcsvax!loomis  
jeff@nosc

Philip J. Mercurio  
Quantitative Morphology Lab  
UCSD M-024  
La Jolla, CA 92093  
(619) 452-3861

ucbvax!sdcsvax!mercurio  
sdcs!mercurio@nprdc

### Abstract

At UCSD's Quantitative Morphology Lab, Unix is used for the control of high performance computer graphics hardware interfaced to a film animation bench. Our animation system has been used to produce scientific films in the areas of neurosciences and linguistics. Several artists have also been turned loose on the system with interesting results.

The presentation will include a brief description of the evolution of the hardware and software and how we took advantage of Unix to save programming effort. We'll also be showing a short film made especially for the conference.

*(Note: What follows is a transcript of the narration to the videotape shown at the conference, and thus is incomplete on its own. To help compensate for this, I have included the textual examples from the videotape here. --PJM)*

---

## History of QMLab

The Quantitative Morphology Laboratory started with research performed by Dr. Robert B. Livingston in the 1960's and 70's. In 1975, an award-winning film called "The Human Brain: A Dynamic View of It's Structures and Organization" was produced. In this movie, a three-dimensional model of the human brain was constructed using computer graphics.

A human brain was first imbeded in plastic. It was then placed on a microtome, which shaved off successive thin layers. The top of the block is photographed at each pass, and these frames may be shown in sequence as a movie. This process is called "cinemorphology". For the human brain film, the frames of a cinemorphology movie were carefully hand digitized, maintaining perfect registration throughout. The digitized sections were then reconstructed into a three-dimensional image using the graphics system at UCSD's Chemistry department, the very first Evans and Sutherland Picture System 1 built. The three-dimensional brain model was filmed using the Chemistry department's computer-controlled camera.

## Hardware

The present system at QMLab consists of a PDP-11/34 and an Evans and Sutherland Picture System II. This system was acquired after the success of the brain film through a grant from Hoffman-La Roche Laboratories. In 1978, Phil Cohen traveled with the system to a number of conferences where participants were able to interact with the three dimensional brain model. Function switches and dials are used to interact with most of the software, although movie making is often more a matter of editing a textual script and then viewing the results.

The camera and filter wheel controllers were built by Robert Abel and Associates and interface to the PDP-11 through serial ports. The camera function box controls a 35 mm Mitchel camera which records an image off of a flat-face CRT made by Xytron. The CRT is slaved to the main Picture System monitor. Each frame is constructed using multiple exposures. Each color is filmed individually through a combination of three filters. Objects which are larger than the Picture System can display at once are exposed part by part. After all of the objects in a frame are exposed the film is advanced.



---

## Software

The software that is used in animation falls into two categories: object generation and motion sequencing. Objects are generated in a number of ways. Some objects are generated using two-dimensional digitizing. The brain is a classic example of this approach. As the operator digitizes an object the picture system displays the data that has already been entered. A three dimensional object is created by stacking two dimensional contours. We have felt somewhat limited by not having general three dimensional object generation software.

Sign language data is also acquired through digitizing. In this case complex three dimensional patterns are captured using infrared cameras. The two clouds of points represent data for the fingertip and the wrist, the bar connects the two points for a given time. The rate at which the bar moves is very important, in a sense the pattern is four-dimensional.

We have also developed software for synthesizing upper arm movements for sign language research. Here the user interacts with the model by manipulating joint angles using dials. After actions have been specified using key frame and tablet techniques they are saved on disk for later replay. The sign language work is being done in collaboration with Ursula Bellugi at the Salk Institute and John Hollerbach at MIT.

Several objects have been generated using mathematical formula. Here a space-filling yinyang curve is being traced out. We have also generated toroids, spheres, dot fields and drum surfaces using this technique.

Finally, we adapt data bases from other sources. For example, all of the fonts in the movie are taken from a data base of the Hershey fonts.

The core of the animation software at QMLab is a program called *movie* written by Roger Sumner in 1978. *movie* is a compiler for an animation language called *moviecode*. Using *moviecode*, a user can construct scene elements consisting of one or more Picture System display files.

```
u = 0;
n = 1;
i = 2;
c = 3;
```

---

```
o = 4;  
m = 5;  
unicom = u n i c o m;
```

Here are some declaration statements taken from the moviecode for the UNICOM logo sequence you'll be seeing later. The first six lines assign names to each of the Picture System display files passed as arguments to the *movie* program--each of these display files contains an image of one of the six letters in UNICOM. The last line defines a new element called "unicom" consisting of the six elements "u", "n", "i", "c", "o", and "m". As you'll see in a moment, this hierarchical definition makes it possible to manipulate each of the six letters separately for the first part of the movie, then manipulate all of the letters simultaneously by referring to the "unicom" element.

```
rotx unicom,360,240;  
roty unicom,-360,210,,30,0;  
scalez unicom,-20000,240,22048;  
scale unicom,-19000,240,20048;  
draw unicom,384;
```

Here are some sample moviecode commands, taken from the latter part of the UNICOM logo movie. The first word on each line specifies what transformation is to be made to the element, and the second word, which element is to be transformed. The numbers following specify how much the element is to be transformed and over what time span. For example, the first statement causes the "unicom" element to be rotated about the x axis 360 degrees across 240 frames, or 10 seconds.

```
define(DUR,12)  
define(END,384)  
define(LETTER,'  
    roty $1,90,0,0,0,0;  
    tranx $1,eval($2-1500),DUR,0,...  
    roty $1,-90,DUR,,eval(END-...  
    $4  
    color $1,240,COLORNO,0;
```

---

We often need to make similar transformations to a number of movie elements. To make this easier, we employ Unix's *m4* macro pre-processor. Here we define an *m4* macro consisting of transformations which need to be applied to each of the six letters in "unicom" separately.

```
LETTER(u,-8000,BLUE)
LETTER(n,2500,RED)
LETTER(i,8750,GREEN)
LETTER(c,15000,GREEN)
LETTER(o,23000,RED)
LETTER(m,32000,BLUE)
```

We then call this macro on each of the six elements. In this case, the first argument is the element, the second argument is a displacement which moves the letter to its proper position in the word "unicom", and the third argument is the color the element should be displayed in.

```
m4 mdef unicom.m | movie -m | color | O > unicom.i
interp -vgs slate list < unicom.i
```

Originally, *movie* was intended to be used both for previewing a movie and for filming it. It later turned out easier to modify *movie* so that it would output a stream of low-level camera control commands and transformation matrices when called with the *-m* (moviemaking) flag. This command stream is first filtered by *color*, a program which adds the low-level commands to control the color filter wheel, then it is passed through *O*, an optimizing filter which reduces the size of the command stream and optimizes filter wheel and camera movements. The resulting command stream is saved in a file for use later. With most of the movies made recently, the moviecode is first passed through the *m4* pre-processor. The camera station hardware is controlled by the *interp* program, which interprets the command stream generated by the pipeline above.

Here is an example of a movie, the UNICOM logo sequence, being viewed in preview mode. Because of the limitations of the Picture System's memory, we are restricted to simpler display files consisting of fewer vectors. In this case, we use single-layered letters rather than the stacked ones you saw earlier. The *interp* program can perform multiple exposures on each frame and can swap display files in and out of the Picture System's memory as needed, allowing images much more complicated than these to be filmed. Also, the Picture System is only capable of displaying

---

black and white images, leaving color for the moviemaker's imagination. For the UNICOM logo movie, we used colored felt markers to help choose the proper color scheme.

Moviecode is often written with the aid of an interactive graphics program called *compose*. Then the moviecode is iteratively edited and previewed until the moviemakers are satisfied. Here we see a second version of the UNICOM logo sequence. An additional rotation has been added to enhance the tumbling in the beginning of the movie.

### QMLab Sampler

What you are about to see is a sampler of films shot here at the Quantitative Morphology Laboratory over the past years. The first piece is about four minutes of graphics taken from the film "The Human Brain: A Dynamic View of It's Structures and Organization" released in 1975. These scenes first demonstrate various structures within the part of the brain called the limbic system, then the entire brain is brought into view as the camera's eye penetrates the surface, providing a view never-before seen of the brain from within.

The next sequence consists of images produced by the three-dimensional analysis of sign language movements being done by Jeffrey Loomis for Ursula Bellugi's laboratory at the Salk institute. These scenes include digital reconstructions of the actual movements of a person signing, and synthesized signs enacted by a stick figure.

The next four sequences are short pieces of animation done by various artists. The first is "On the Road Again", by Phil Cohen and Roger Sumner, the second is "Dance", taken from the work of Pat Gallagher, the third is "Zoom", by Phil Mercurio, and the fourth is "Yin Yang", by Jeffrey Loomis.

Following these are some images of two Unix networks. The first demonstrates the interconnections between the Unix sites at the campus of the University of California at San Diego. The second shows a map of the United States with arcs connecting the various Usenet sites. The final piece is the UNICOM logo sequence. These last two films were created by Jeffrey Loomis and Phil Mercurio. The music is by the Grateful Dead.

---

Friday, 29 Jan 1983

Session: F1B — New UNIX Implementations II

Chairperson: *Joel Carter*

## **REGULUS, a Real-Time UNIX Lookalike**

*Bill Allen*

Alcyon Corporation  
8716 Production Avenue  
San Diego, CA 92121

REGULUS is source level system call compatible with UNIX. All UNIX system calls are implemented and have the same names, same arguments, and same return values. REGULUS supports standard UNIX features: multiuser, multitasking; tree structured file system; device and file independent I/O; pipes and filters; runtime I/O redirection; shared code programs; and a full complement of software development tools.

REGULUS has many extended capabilities. Real-time tasks can be locked in memory, preemptively scheduled by priority, and prevented from being time-sliced. Inter-task communication includes 32 'user' signals which have no default meaning to the system; the default action is ignored.

A large improvement went into the design of the file system. These include dynamic allocation of file index records, fewer disk accesses for sequential data, and index records stored next to file data blocks. Tasks can lock multiple file segments as well as choose the response to lock failure. The file segment can be locked for reading, writing, or both.

One of the big advantages of REGULUS is the minimal hardware support required. Only 128kb of RAM is needed for a complete system. With one megabyte of disk storage and an OEM—provided dedicated kernel as small as 32kb REGULUS requires very minimal hardware.

---

## UNIX for the National 16032

*Paul Neelands*

*Richard Miller*

*and Chris Sturgess*

Human Computing Resources Corporation  
10 St. Mary Street  
Toronto, Canada M4Y 1P9

The NS 16032 is a true 32-bit processor. Although its current hardware interface to the outside world is via 16 multiplexed bits, the instruction set was designed from the start to look like a big machine. It has VAX-like memory management unit and also supports page reference bits. Full floating point support is available. The 16032 instruction set allows high density code. The machine provides support for high level languages such as Pascal and C without being restrictive.

The base hardware for the porting effort was a prototype workstation with the 16032 CPU, the 16082 memory management unit, and the floating point chip. The machine had 256kb of main memory, a Winchester disk, a streaming tape drive, a GPIB interface, RS232, a bit-map display, and limited audio output.

The C compiler for this effort was based upon the VAX version of the portable C compiler. The assembler and loader were difficult because the machine has a complex instruction set.

The UNITY kernel for the 16032 is initially based upon Berkeley 4.1BSD. This was pulled off with only 256kb of main memory! Much grunge and a number of machine dependencies had to be removed. Added were multiwindow graphics support, audio output support, and, in the future, GPIB support. The major problem in the porting effort was to circumvent minor hardware bugs in the early versions of the chips.

The system was first demonstrated in public at the November 82 Comdex meeting. Future plans call for:

- first OEM versions in March of this year
- a full rewrite of the demand paging algorithm
- unification of paging and the buffer cache into one concept
- increased standardization of the utility programs in order to conform to industry standards
- network support
- move the enhanced demand paging code back to company's VAX product.

---

UNIX for the  
National 16032

Human Computing Resources Corp.  
10 St. Mary Street  
Toronto, Canada M4Y 1P9  
416-922-1937

EXTENDED ABSTRACT

This is an extended abstract of a talk to be presented at the January 1983 Unicom meeting in San Diego. The talk describes an implementation of the UNIX system for the National Semiconductor 16032 32-bit microprocessor. This work was performed by

Richard Miller  
Paul Neelands  
Chris Sturgess  
Tracy Tims

and other members of the HCR staff.

The NS 16032 is a true 32-bit processor. Although its current hardware interface to the outside world is via 16 multiplexed bits, the instruction set was designed from the start to look like a big machine. It has VAX-like memory management unit. It fixes the worst flaw of the VAX MMU, since it does support page reference bits. Full floating point support is available. The 16032 instruction set allows high density (code better than the VAX). The machine provides support for high level languages such as Pascal and C. Unlike some other machines, it does this without being restrictive. (Some machines are determined to be

-----  
UNIX is a trademark of Bell Laboratories. UNITY is a trademark of Human Computing Resources Corp. VAX is a trademark of Digital Equipment Corp.

---

Pascal or Ada machines, much to the detriment of a C compiler.) In general, the machine is "done right". It is very well suited to UNIX.

HCR has developed a version of its UNITY operating system (an AT&T licensed derivation of true UNIX) for this machine. The talk discusses some of the issues in porting UNIX to this machine.

The base hardware for the porting effort was a prototype workstation with the 16032 CPU, the 16082 memory management unit, and the floating point chip. The machine had 256 KB of main memory, a winchester disk, a streaming tape drive, a GPIB interface, RS232, a bit-map display, and limited audio output.

The C compiler for this effort was based upon the Vax version of the "portable" C compiler. The uniform instruction set made the job relatively easy. The assembler and loader were in fact more difficult, because the machine has a complex instruction set. The linkage editor was complicated because of the variety of relocation types and addressing modes. A notable feature of the machine is the "module table", which allows replacement of modules without recompilation or relinking. Our software in fact uses a more traditional approach with a standard linkage editor, although we make use of the module table in order to reduce code size.

The UNITY kernel for the 16032 is initially based upon Berkeley 4.1BSD. (Note that we pulled this off with only 256 Kb of main memory.) We had to remove much grunge and a number of machine dependencies. Otherwise the kernel work was fairly straightforward. We have added multiwindow graphics support, audio output support, and we will have GPIB support. The major problem in the porting effort was to circumvent minor hardware bugs in the early versions of the chips. (Note: all of the early hardware flaws are handled transparently and efficiently by the kernel -- none are visible to user level code.)

UNIX utilities are very easy to move from the VAX. It is very hard for a program to tell that the machine is not a VAX, since it has 32-bit words, 32-bit byte pointers, a downward-growing stack, the same char order within short ints, the same short int order within long ints, 32 and 64 bit floating point, and signed chars.

The 16032 supports full demand paging. All instructions can be backed up upon a page fault. The MMU is essentially like the VAX. We used the Berkeley kernel as a base, since it was the only available paging kernel. However, Berkeley 4.1 has many drawbacks. Good demand paging is important for small



---

workstations with smaller amounts of main memory and slower disks. The Berkeley kernel falls down at two interesting end conditions: If one runs a single large job bigger than main memory, the 4.1 code is VERY slow, because it has a fixed page-out rate, and it will sit doing nothing rather than freeing up more page frames. On the other hand, if one runs many small jobs (i.e. a typical heavy UNIX timesharing load) then the 4.1 kernel also performs poorly, since it has no idea of a per-process working set. We have made a number of improvements to the algorithm, and we expect to replace it entirely in the very near future.

The multi-window graphics support is also of interest. Our initial development machine was equipped with a full bit-map display. We have brought up an all software implementation of bitmap graphics (i.e. we don't use any special "rasterop" hardware). The 16032 is quite good at this, especially since it supports bit addressing.

This system was first demonstrated in public at the November 82 Comdex meeting. We expect the first OEM versions in March of this year. Our future plans call for a full rewrite of the demand paging algorithm, unification of paging and the buffer cache into one concept, increased standardization of the utility programs in order to conform to industry standards, and network support. Finally, we expect to move the enhanced demand paging code back to our VAX product.

---

# The Port of UNIX to the Gould 32/27

*Jack Blevins*

Gould, Inc.  
S.E.L. Computer Systems Division  
Post Office Box 9148  
Fort Lauderdale, FL 33310-9148

The porting effort began in October, 1981 with a team of five programmers. The Gould S.E.L. MPX operating system was the development environment. Two unique features of the Gould S.E.L. hardware are an 8192 byte page and an I/O system that uses a 26.6 Mbyte per second SelBUS and a 1 Mbyte per second MPBUS. The SelBUS interfaces high speed peripherals like disc, tape, and memory while the MPBUS interfaces terminal controllers, floppy discs, line printers, and communication controllers. An I/D Processor "IDP" connects the MPBUS to the SelBUS.

The actual port was performed from UNIX Version 32. Version 32 was selected over Version 7 because it had been converted to a 32-bit processor. The portable C Compiler had previously been ported to Gould S.E.L. computers.

Armed with these tools *mfks*, *fboot*, *tboot*, *ldcopy*, and the kernel were tackled. Naturally many obstacles had to be overcome. The team learned about *Ld* and the difference between the object code generated by the assembler and that expected by *Ld*. A C language program called *oc* was developed to convert the object code format and port *Ld* to MPX.

The first kernel was linked and tested in early February of 1982. An MPX debugger was linked into the kernel to provide the means to determine the errors and possible patches for them. This debugger was only useful for one person testing at a time.

In late May 70 utilities were operational and the system was stable as long as there was no swapping of user programs. The swapping problem was corrected by mid-June and a very stable UNIX was usable by the development team and software quality assurance.

The source code received from Bell Laboratories had no major bugs in the operating system, and the gamble on the code from Bell paid off. The compiler errors could be avoided by changing C source. The utility programs were the major source of porting difficulties because differences in code and documentation were found in 25% of the utilities.

All in all, the package supplied from Bell Laboratories was good, as demonstrated by the fact that a team of five and later nine could port it with no outside assistance.

---

## The Port of UNIX to the Gould 32/27

Jack Blevins  
Gould Inc., S.E.L. Computer Systems Division  
Fort Lauderdale, Fl. 33310-9148

### 1. Abstract

A team of five ported UNIX to the Gould S.E.L. 32/27 computer. The initial effort occurred under the MPX operating system with the completion of the port using UNIX. About 120 of the utility programs were ported without the benefit of a debugger. The source code provided by Bell Laboratories is rated good while the documentation is rated fair. The port lasted from October, 1981, to August, 1982, while the quality assurance testing took place from July, 1982, to December, 1982.

### 2. Introduction

Gould S.E.L. Computer Systems Division designs, manufactures, sells, and services 32-bit super minicomputers. Our traditional markets have been real time or CPU intensive. We have been manufacturing 32-bit minicomputers since 1969 and we are currently one of the fastest growing minicomputer firms in the nation with revenues approaching \$200 million.

The decision to port UNIX to our computers was made for two reasons. First, our existing customer base needed a better program development environment. Second, we wanted to open new market segments but our efforts were thwarted by not having a widely user acceptable operating system. The cost of software development has become such a large portion of system cost that new customers were wary of changing to an operating system supported only by a single vendor of our size. (Note that real time applications still rely upon our real time operating systems.)

### 3. The Hardware

The initial port was to our Concept 32/27 computer with two 80 Mbyte disc drives, a single magnetic tape drive, an upper/lower case line printer, and sixteen terminals. The central processor and device controllers are all designed by Gould S.E.L. while the peripherals are purchased.

---

Since none of us had ever been a UNIX user and only two had been with Gould S.E.L. before this project, we undoubtedly made some mistakes an experienced team would not make. We quickly learned about 'ld' and the differences between the object code generated by our assembler and that expected by 'ld'. We developed a 'C' language program called 'oc' to convert the object code format and ported 'ld' to MPX. This gave us the tools to build UNIX.

While these tools were being developed, the hardware dependent portions of the kernel were being replaced. The modules 'locore' and 'trap' were rewritten and a new module 'lobild' developed to provide all hardware specific functions except those for device controllers. The device drivers were left intact as much as possible to minimize the knowledge needed about the I/O features of the kernel. A module 'selio' was written to emulate the CED device controllers and drive our controllers. Finally, the modules 'bio', 'slp', 'sysl', and 'text' were modified to support our memory mapping hardware.

The file system structure was not changed and was expected to be error free because of the extensive file system use within UNIX. Operating under MPX the modified 'mkfs' program built a proper file system and wrote the executable modules to it.

The first kernel was linked and tested in early February. An MPX debugger was linked into the kernel to provide the means to determine the errors and possible patches for them. This debugger was only useful with one person testing at a time. By mid-March '/etc/init' would load, execute, and bring in 'sh'. By April 'ed' and a few simple utilities would run. In late April the 'C' compiler would run. At this point we stopped using MPX and completed the development under UNIX.

Around the first week of April three more programmers were added to the staff. All of these people were new to Gould S.E.L. and one had some previous UNIX experience.

One of the earliest decisions of the project was to deliver the documentation in the same form as received from Bell Laboratories. A technical writer joined the team and prepared the documentation for printing. It was soon decided to reorganize the manuals and supply a better table of contents. As corrections to the documentation were developed they were applied. The technical writing group determined that the long term future of the documentation included a total reformatting and rewrite as necessary.

The lack of knowledge about the hardware I/O system by the development team led to operating system stability problems. First we were improperly blocking and unblocking interrupts which would cause us to clear some interrupts without actually servicing them.

---

Two unique features of our hardware are an 8192 byte page and an I/O system that uses a 26.6 Mbyte per second SelBUS and a 1 Mbyte per second MPBUS. The SelBUS interfaces high speed peripherals like disc, tape, and memory while the MPBUS interfaces terminal controllers, floppy discs, line printers, and communication controllers. An I/O Processor (IOP) connects the MPBUS to the SelBUS.

The operating system is mapped into the first thirty-two pages with physical addresses matching logical addresses. A user process may acquire up to thirty-two of the 8-kbyte pages which are then mapped to logically follow the operating system. The operating system structures for the process are contained in a page mapped into the operating system. The maximum memory size for UNIX is four Mbytes.

Our processor does not have a stack so one is emulated using a general purpose register.

#### 4. The Bell Laboratories Software

The actual port was performed from UNIX Version 32. Version 32 was selected over Version 7, because it had been converted to a 32-bit processor. The portable 'C' Compiler had previously been ported to Gould S.E.L. computers by Bell Laboratories and was purchased separately from UNIX.

When UNIX System III was made available we purchased a source License for the VAX version.

#### 5. The Port

The porting effort began in mid-October 1981 with a team of five programmers. The Gould S.E.L. MX operating system on the hardware described earlier was the development environment.

Although we had a 'C' compiler that generated assembler source for our computers, we did not have a run time library to link with the compiler and execute it under MPX. We developed a rudimentary run time library in about three weeks. As soon as the 'C' Compiler was up, we recognized the MPX editor was not adequate and 'ed' was ported.

Armed with these tools we tackled the 'mkfs', 'fboot', 'tboot', 'tdcopy', and the kernel. Since the goal was to ship UNIX in a year, we made some optimistic assumptions that determined our approach to the port. These were 1) the compiler would generate valid code; 2) all areas of the kernel would be stable; and 3) we could maintain exact functionality with Version 32. Thus we began the port adding new modules as the need arose.

---

Since our system was unstable we seemed to spend more time recreating the file system than doing development. The "-s" option of 'icheck' failed and we had to perform a 'tar' of the file system to rebuild it. 'Icheck' was fixed later by assigning a short to an integer before doing a compare. The problem with 'icheck' was traced to a compiler error which caused comparisons using shorts to be done incorrectly.

While the system was soon stable in single user mode, the multiple user mode was still unstable. these problems were traced to using the clock as the highest priority interrupt. Over a two week period various fixes were applied until stability was achieved.

In late May we had 70 utilites operational and a stable operating system as long as there was no swapping of user programs. The 'C' compiler was found to have a few annoying bugs. Among these were unary minus on doubles failed and compound operators for multiplication and division produced code that would not assemble. All problems were corrected by writing the offending statement in a different manner. Then we gave software quality assurance its first copy of UNIX.

The swapping problems were corrected by mid-June and a very stable UNIX was usable by the development team and software quality assurance. All of June was spent bringing up the rest of the utilities. We ported everything except FORTRAN and its related tools, the communication software, and the graphics software. The two debuggers would not port without major changes so they were left for the next phase. (The lack of a debugger was a major challenge while porting the utilities.)

The ninth member of the development team came on board in late June with no professional experience as he had just graduated from college.

The 'C' compiler was enhanced to support shared text and large, often used utilities were recompiled to use it. The processes functioned correctly but some new swapping problems were introduced. These were fixed one-by-one and by early August the system with shared text was stable for general use.

## 6. The Software Quality Assurance

The formal software quality assurance acceptance of the UNIX product began in late July. 'Sh' scripts had been developed during the UNIX port and were used to test the utility programs. The documentation underwent scrutiny and many discrepancies were discovered between it and the program execution. These were either fixed in the utility program or mention made in the "bugs" section of the documentation.

---

The software quality assurance testing concluded the first week in December and the product was ready for shipment. Software quality assurance also indicated that the UNIX documentation should be brought up to corporate standards.

## 7. The First Demonstrations

The early quality assurance testing of UNIX went well, and by late September it had been demonstrated in Paris, France and at the Federal Computer Conference in Washington, D.C. After those shows, the disk I/O performance was improved by increasing the disk sector size to 1024 bytes.

## 8. Summary

The source code received from Bell Laboratories had no major bugs in the operating system, and our gamble on the code from Bell Laboratories paid off. The compiler errors could be avoided by changing 'C' source. The utility programs were the major source of porting difficulties because differences in code and documentation were found in 25% of the utilities.

All in all, the package supplied from Bell Laboratories was good, as demonstrated by the fact that a team of five and later nine could port it with no outside assistance.

\*UNIX is a trademark of Bell Laboratories.

---

# A Menu-Driven Real-Time UNIX System

*Kent Blackett*

MASSCOMP  
543 Great Road  
Littleton, MA 01460

A laboratory computer system based on UNIX was developed which can reliably and predictably achieve two Mbytes per second throughout with interrupt latencies as low as 2 micro-seconds. The system is based on UNIX system III and augmented by UCB's 4.1BSD. The real-time extensions that were developed were:

- fixed priority schedules
- memory locked processes
- contiguous disk files
- priority-delivered user interrupts
- high performance pipes

A high speed bipolar front end processor offloads the host. Modifications to UNIX improve system throughput and file integrity. Layered software was added to provide user ease-of-use through window management and "quick-choice" menu selections.



---

## A Menu-Driven Real-Time UNIX™ System

Kent Blackett  
MASSCOMP  
543 Great Road  
Littleton, MA 01460

MASSCOMP began the development of the MC-500 32-bit computer system with a set of software development goals. First, in keeping with the industry standard nature of the processors (68000 chips) and the buses (MULTIBUS™ and STD Bus) we would use an industry standard operating system. This meant UNIX, or a UNIX-like system, since we did not feel CP/M was adequate. The second goal was that we should make the system easy-to-use as well as easy-to-learn. We also wanted to make the machine usable with little or no programming. Our third goal was to provide high-speed data acquisition, high-performance graphics, and graphing. And lastly, we also wanted to make the machine a very capable software development system.

We decided to use UNIX System III™ from Bell Laboratories as the base operating system for the machine. We add to System III several key pieces of the operating system from the University of California, Berkeley, distribution of VAX™ UNIX. For example, much of the virtual memory management and several new utilities are derived from the Berkeley distribution. Because UNIX is a high-performance time-sharing system but not a high-performance real-time system, several MASSCOMP developed real-time components were added to the base operating system.

UNIX is a trademark of Bell Laboratories.  
MULTIBUS is a trademark of the Intel Corp.  
SYSTEM is a trademark of Bell Laboratories.  
VAX is a trademark of the Digital Equipment Corp.

---

Sub-millisecond event handling is handled by a dedicated bit-slice processor because a multi-user operating system is not able to provide sub-millisecond response to real-time events. Moreover, the dedicated processor, called a Data Acquisition and Control Processor (DA/CP) shields the user from the indeterminacies of varying system loads. Although the use of a dedicated real-time processor offloads extremely time-critical operations from the main CPU there is still a need for the operating to offer real-time capabilities. MASSCOMP has enhanced UNIX with a number of powerful real-time capabilities.

As a very capable time-sharing system, UNIX juggles the priorities of the processes that it is currently executing (based on their recent execution history) to give the best response to its time-sharing users. The very nature of this type of adjustment introduces an unpredictable nature into the execution profile of each program. This loss of predictability is at direct odds to good real-time performance.

MASSCOMP added a fixed priority scheduler with which the operating system will not interfere, so that the user can assign priorities. When a real-time process needs to run, it will run until it decides that it would like to relinquish the processor, or until a higher priority real-time event occurs.

We have also added the ability to lock processes in memory so that real-time processes neither participate in demand paging nor in swapping, since disk accesses in the middle of process execution would introduce a large measure of unpredictability.

MASSCOMP added continuous disk allocation (which Bell Laboratories' UNIX doesn't support) because if you're taking data at a specified rate and trying to get it out to the disk, you can't afford unexpected disk seeks in the middle of your data files.

---

We've also put in a capability found in other real-time operating systems called Asynchronous System Traps (AST's). AST's allow the programmer to schedule subtasks of his own process to run at a software priority level. Thus, external and internal events can cause tasks to be executed.

The UNIX operating system enables the user to write programs as a number of small independent processes which pass data down a Pipe from one process to the next. In keeping with the UNIX philosophy, and knowing that a lot of our real-time data acquisition analysis and graphics would depend on Pipelines, we have improved the performance of this key system feature. UNIX was originally written for very small address machines which means that it is quite stingy in its use of memory. We've been able to take advantage of our large virtual addressing space and couple that with a couple of other changes to the way the Pipes work, to produce a higher performance Pipe system.

A key part of the user interface system is that we view each Independent Graphics Processor (IGP) as a multiple window, or multiple virtual terminal, device. On a single graphics screen we present a number of windows, or viewports, each one of which is running an independent process on behalf of the user. Instead of running one or two large programs sequentially, we run simultaneously several small programs for the user. Each window, or virtual terminal, on an IGP is tied to one (or more, if they've been Piped together) processes.

All interaction with the machine is gathered through a menu-driven system (called Quick-Choice), which answers a lot of the requirements for an easy-to-learn system. The key to a menu-driven system is that at any point in time it presents to the user a constrained list of alternatives from which the user may choose his next command. The

---

next frame of the menu system presents him with a constrained set of alternatives from which to make his next choice. The user moves down what is conceptually a tree structure until the job that he wants to do is completely specified.

Ease-of-use was a system goal as well. Once the user has used our system and finds himself repeatedly using the same path in the menu system he can say, "Gee, I seem to be doing this a lot. I will instruct the system to learn this path through the tree system and store the path away as a separate procedure." Therefore, the next time he uses the machine he need only enter a single command (or press a single key) which will cause the entire menu-traversal (as we call it) to be invoked.

Software which is written by the user can easily be integrated into the menu interface so that user-supplied programs are as easy to use as MASSCOMP-supplied modules. The combination of the User Interface plus real-time capabilities enables MASSCOMP to offer a uniquely powerful version of UNIX.

---

# EUNICE

*Ellen Williams*

The Wollongong Group, Inc.  
1135A San Antonio Road  
Palo Alto, CA 94303

EUNICE is software that provides the capability to run UNIX programs under VMS. This gives VAX users both UNIX and VMS on the same machine. They can operate in either environment with all facilities of the other immediately accessible. The user can enter file specifications in either UNIX or VMS forms. UNIX programs running in the EUNICE environment can read and write both VMS and UNIX files, and read VMS directory files as though they were UNIX directories. When using UNIX compilers under EUNICE the user can generate either VMS or UNIX object files. VMS object files may be linked with other modules from other VMS languages.

VMS users benefit from additional features and software available through EUNICE/UNIX, including:

- Escape to the UNIX environment

- Any of the more than 200 UNIX utilities

- High level languages, such as C, FORTRAN 77, RATFOR

- Compatible file, device and interprocess I/O

- Shell command language (*sh* and *cs**h*)

- I/O re-direction

- Pipes for interprocess communication

- Command chaining

- Networking

- Program development tools:

  - Program maintenance system

  - Compiler-compiler

  - Lexical analyzer

- Text processing and document preparation software:

  - Full screen and line editors

  - Formatter for correspondence quality output

  - Phototypesetter formatter

  - Preprocessor for mathematical equations

  - Preprocessor for tables

- Communications software

- On-line documentation

The EUNICE package is available for all VAX/VMS systems (11/730, 11/750, 11/780, and 11/782).

---

Friday, 29 Jan 1983

Session: F2A — New UNIX Implementations III

Chairperson: *Mike O'Brien*  
Rand Corporation

## Porting UNIX

*P. Verbaeten and Y. Berbers*

(main authors)

*("A. Wupit" collective name of the participants of the TWIX project)*

Katholieke Universiteit Leuven  
Dept. Computerwetenschappen  
Celestijnenlaan 200A  
B-3030 Leuven  
Belgium

In our department we have ported the UNIX V7 computer system, and have gained much experience about many different aspects of porting an operating system to a new hardware environment, especially about porting UNIX.

The emphasis here will be on the porting strategy and the working environment; we will briefly discuss some alternatives and the major problem areas of a UNIX port.

---

## PORTING UNIX

A. Wupit (\*)

Computer Science Department,  
Katholieke Universiteit Leuven, Belgium

### 1. Introduction

In our department we have ported the UNIX V7 (\*\*) computer system, and have gained much experience about many different aspects of porting an operating system to a new hardware environment, especially about porting UNIX.

The emphasis here will be on the porting strategy and the working environment; we will briefly discuss some alternatives and the major problem areas of a UNIX port.

### 2. Strategy

#### 2.1. Working environment

We have chosen to work under a running UNIX system, and to download code to the target machine.

Advantages of this approach are :

- a familiar working environment,
- use of existing software : The C-compiler for the 68000 is a modified version of pcc; the assembler was written using yacc; the loader is based on the standard UNIX loader. Lint was very valuable to us as we had many type problems in the C-code (our C-compiler has 16-bit integers and 32-bit pointers as opposed to 16-bit integers and 16-bit pointers on the PDP 11).

As disadvantages we mention crosscompilation and downloading.

Alternatives include working with an existing operating system on the target machine (with disadvantages such as shutting down and starting up the operating system at each test, working with two different operating systems) and a combination of the two previous possibilities : the programmers work bench.

---

(\*) A. Wupit is the collective name of the participants of the TWIX project, a joint effort between the "Departement Computerwetenschappen" of the "Katholieke Universiteit Leuven" and the "Unité d'Informatique" of the "Université Catholique de Louvain". The main authors are P. Verbaeten and Y. Berbers.

(\*\*) UNIX is a trademark of Bell Laboratories

---

Some debugging facilities on the target machine are necessary (breakpoints, display and modification of registers and memory, etc.). A virtual console with a monitor in ROM is a valuable tool here. A printer connected to this console is very useful as debugging messages in the system have been used frequently.

## 2.2. Building UNIX file systems on the target machine

The most obvious solutions are working with compatible media (with UNIX on another machine) or using special software on the target machine.

We have chosen for a communication link (with a packet driver) between the target machine and UNIX on another machine : I/O requests for disk are sent to the UNIX system and executed there on a special file system.

There are several advantages to such a remote file system :

- the file system can be build under UNIX (no additional software at all),
- once we had a real disk we mounted a local and a remote file system; copying of sources was then achieved with normal UNIX commands!

The only problem encountered dealt with the different byte addressing on the PDP 11 and 68000.

## 3. Porting steps

1. Adapting the running UNIX system so that it no longer makes use of hardware features which do not exist on the target machine. This modified UNIX is then first tested on an existing UNIX machine. The advantages of this approach can be twofold :

- the facilities on a running UNIX system are usually much better then the ones available on the target machine,
- after the simulation, the strict porting can be started with correct system code, which makes testing much easier.

We did this for the missing virtual addresses in kernel mode.

2. Partial bottom-up testing : This was done for all assembler routines and a few C-routines. Building complete test environments for all routines to be tested is however a cumbersome task.

3. Top-down testing of the rest of the system. We started with a very scaled down version of the system (no clock, no I/O devices, etc.) and gradually added more features.



---

We mention a few substeps :

- startup of the first user process; the code of it is included in the system code,
- handling of traps and the system call mechanism,
- disk I/O via the packetdriver,
- some file I/O,
- the fork and exec system call.

For all these steps, every test requires downloading of the operating system, as the code for the user process is included in the system. But the system is still limited at this stage, so downloading does not take a long time.

At this point all facilities are available to start the execution of a program residing on disk (cfr. /etc/init). Now several tests can be done with the same downloaded system.

We proceeded with :

- the remaining system calls,
- terminal I/O.

At this stage we can easily build a very restricted shell and proceed with the next two steps (in parallel if one so desires). The kernel should be OK now.

4. Porting of the software tools; the most important among these are the shell, the editor, the compiler, assembler and loader.
5. Installation of a disk and diskdriver and transferring the complete file system (see above, point 2.2).

#### 4. Conclusion

##### 4.1. The portability of standard UNIX

An operating system is never hardware independent. But as UNIX is written for 90 percent in C (a high level language), it is reasonably portable.

The main problem areas are :

- the routines written in assembler : about 10 percent of the code,

- 
- hardware dependent constants : most of these are grouped in header files,
  - memory management : very machine dependent but quite local,
  - handling of interrupts and traps : very machine dependent but quite local,
  - devices : very machine dependent but quite local,
  - type problems (these problems exist because of the weak type checking done by C) : not local at all, dispersed throughout kernel and tools. Lint is very valuable here.

#### 4.2. Improvements

At this moment our department has at its disposal a single set of very machine independent C-routines (routines in the kernel and utilities) which run on both PDP 11 and 68000.

Starting from this portable version of the UNIX kernel on the PDP 11, only a small number of the 29 header files and 227 C-routines were modified so that UNIX would run on the 68000 : changes were made to 6 header files and 14 C-routines and 2 new routines were added.

Starting from the "standard" UNIX V7, 9 header files and 33 C-routines were modified and 2 new routines were added.

---

**This page intentionally left blank**

---

# UNIX on the National Semiconductor NS16032

*Glenn C. Skinner*

National Semiconductor  
Microcomputer Systems Division  
M/S 7C-266  
1135 Kern Avenue  
Sunnyvale, CA 94086

*Bill Jolitz*

Symmetric Systems  
21467 Aldercroft Heights  
Los Gatos, CA 95030

We have recently ported UNIX to the NS16032 microprocessor. The port is based on the highly successful Berkeley VAX system and has been tailored to the microprocessor environment. The NS16000 chip set family provides hardware support for full demand-paged virtual memory, modular software, runtime debugging, and high-performance floating point. These features are well-suited to UNIX and greatly speeded the porting effort; it took a month to get a "bare bones" kernel running, another month to add protected virtual address spaces, and only an additional month to implement demand paging.

These features have been exploited to provide a high quality software environment for the design and construction of large software products. This system (both hardware and software) allows for very low cost computers with state-of-the-art software.

The kernel is designed to exploit fully the virtual memory architecture of the NS16000 family. Each process runs in a protected linear virtual address space of up to 16MB. Page tables for a full size process occupy 129KB, a substantial fraction of the system's physical memory. To reduce this mapping overhead, the kernel can page user process page tables (both first and second level) as well as user process text and data pages. The kernel can also page parts of its own text and data spaces. Thus, large address spaces can be supported on much lower cost machines than was previously possible.

The kernel provides additional virtual memory features. There are new *vspy* and *vlock* system calls for mapping parts of one address space into another and for locking pages into physical memory. These features support time-critical applications by allowing processes to take direct control of device registers and buffer areas. The kernel virtual memory implementation is also designed to support memory-mapped files, explicitly shared segments, and copy-on-write shared segments.

In addition to virtual memory, the NS16032 supports modular software. The processor includes several registers that mediate access to local data, to function and procedure arguments, and to references to external data and routines. This organization has several implications for UNIX. Code size is reduced, since absolute 32-bit addresses can be replaced with small local offsets. Relocation and loading are simplified; code itself need never be modified when loaded and can easily be made ROM-resident (one intriguing possibility is a ROM-resident, dynamically configurable UNIX kernel). However, object files no longer consist of homogeneous text and data segments; instead these segments are partitioned into modules. One of the more challenging parts of the port has been to devise an *a.out* format that captures module-related information without violating existing UNIX conventions on address space organization.

A major aspect of the NS16032 UNIX system is its hardware debugging features. The memory management unit supports two breakpoint on reference address registers and limited program flow tracing. The system has been enhanced to support these features, significantly easing process debugging. For example, one can locate code that might be destroying the contents of a given data cell by trapping writes to that cell. Similarly, a better picture of a program's abnormal behavior may be found by recording its last few (possibly conditional) jumps with the program trace registers. A major step of the port has been to implement a UNIX version of *ddt*

---

that grants user access to this enhanced debugging environment. We have found it invaluable for interactively debugging the kernel itself as well as for the more prosaic purpose of debugging user programs.

Finally, the system has a fast floating point instruction set with addressing modes identical to those of the standard instruction set. This instruction set symmetry makes for straightforward and efficient code generation by compilers; no special addressing cases need be considered merely because one or more operands are floating point.

---

## UNIX† on the NS16032

*Bill Jolitz*

Symmetric Computer Systems  
Los Gatos, California

*Glenn Skinner*

Microcomputer Systems Division  
National Semiconductor Corporation  
Santa Clara, California

### ABSTRACT

We have recently ported UNIX to the NS16032 microprocessor. The port is based on the highly successful Berkeley Vax system and has been tailored to the microprocessor environment. The NS16000 chip family provides support for full demand-paged virtual memory, modular software, runtime debugging, and high-performance floating point. These features are well-suited to UNIX and greatly speeded the porting effort; it took a month to get a bare bones kernel running, another month to add protected virtual address spaces, and only an additional month for demand paging.

Each process runs in a protected linear virtual address space of up to 16MB. The kernel can page user process page tables as well as ordinary user process pages. The kernel can also page parts of its own address space. The chip set's modular software features allow absolute 32-bit addresses to be replaced with small local offsets. Relocation and loading are simplified; code itself need never be modified and can easily be made ROM-resident. The text and data segments of object files are partitioned into modules to support this. Debugging features, supported by the memory management unit, include two breakpoint on reference address registers and limited program flow tracing. A major step of the port has been to implement a UNIX version of *ddt* that grants user access to these features. Finally, the system has a fast floating point instruction set with addressing modes identical to those of the standard instruction set. This instruction set symmetry allows straightforward and efficient code generation by compilers.

### 1. Introduction

To support program development on National Semiconductor Corporation's (NSC) Microprocessor Development System for the NS16032, we decided to port the UNIX operating system to the NSC hardware. The choice of UNIX hardly needs explanation. For us, a more interesting question was which variant of UNIX to port. Since our microprocessor hardware is able to support high performance

---

†UNIX is a Trademark of Bell Laboratories.

---

virtual memory and floating point, we wanted a version that would let us fully exploit these features. After examining possible candidates, we decided to port the Berkeley 4.1bsd VAX UNIX system. This system attracted us because it has a high performance filesystem, virtual memory support, and a variety of existing software systems (e.g., Caesar and VAXIMA) that could enhance our microprocessor's appeal. Other candidates such as System III and Version 7 simply did not mesh as favorably with our hardware.

Our strategy in this UNIX port was to adapt the system wherever possible to take advantage of hardware features but while still remaining as close to 4.1 as possible. To be precise we wished to exploit hardware features of memory management, modular software, and software debugging (both supervisor and user programs). We chose to remain standard by either making a given feature totally invisible to user processes or by isolating the feature to an area as yet untouched by current 4.1 software. Thus the main body of 4.1 functionality remains constant and unchanged, while our new software may take advantage of features that are uniquely provided by the NSC hardware. National 16032 UNIX is upwardly compatible with 4.1bsd.

In the remainder of the paper we start by describing our target hardware, with special attention given to the 16000 chip set. In section three we note several areas where our system differs from 4.1bsd. Section four details the history of our porting effort. In section five we sketch our plans for further system development.

## **2. The Target Hardware**

This port of UNIX has required considerable effort, as it is a port to a new microprocessor with a new architecture. Moreover, all of the surrounding hardware is unique, from the signal bus to each of the peripheral controllers. Very little software for any of the hardware existed at the beginning of the port, and most of that was still under development and quite bug ridden. Similarly, the hardware (silicon most definitely included) was under development during the porting process. In many cases there were long delays waiting for various pieces of equipment to be ready before the port could proceed. For example, the C compiler began generating test programs at a time when the microprocessor had only half of its instructions functioning.

The remainder of this section first sketches out our hardware configurations at the system level and then proceeds to a more detailed description of the 16000 chip set.

### **2.1. Target System Description**

Our porting work began on one internal product, a single-user workstation, and has continued to another product, a multi-user timesharing system.

#### **2.1.1. Workstation**

Our original hardware configuration is a single-user workstation, consisting of a 608 x 800 pixel bitmapped screen and keyboard, with integral 8" winchester and streaming tape controllers and 256KB of memory. An IEEE-488 interface is used as a high speed interface for file transfer from our host VAX 11/780. The VAX has a multidrop fileserver which greatly aided us in supporting the early UNIX environment.

---

### 2.1.2. Timesharing System

The timesharing system consists of much the same hardware as the workstation. The differences are as follows. The bitmapped display and keyboard and associated controller board are not included. Replacing them are one or more 1MB extension memory boards and an eight port serial i/o (RS-232) board. The system packaging is also different, as there is no need to accommodate the display.

## 2.2. The NS16032 Central Processing Unit

The 16032 is a two address processor. It has 32-bit data paths and a 16-bit multiplexed data bus. The 16032 can directly address up to 16MB with its 24-bit wide address bus. Instructions comprise a variable sequence of bytes, from as few as one byte to as many as 24 bytes. There are signed and unsigned byte, word, and double word numeric data types. These data types must be byte-aligned but do not require more stringent alignment. There are also bit and bit field data types, which have no alignment requirements.

### 2.2.1. Register Set

The 16032 processor has a general register set of eight 32-bit registers. None of these registers serve any special purpose for the processor. Each register may be used for holding either 8, 16, or 32-bit data items, as well as for addresses.

The general purpose registers really are general purpose, because the processor also includes eight special registers that handle chores assigned to "general purpose" registers in many other architectures. The program counter, stack pointer, and frame pointer registers function as their names suggest. There are also addressing modes (see below) that specifically use these registers. The processor status register contains condition codes and processor control information. The static base and module registers contain information pertinent to the currently executing module (see below). The interrupt stack register is an alternate stack pointer that can be used instead of the normal stack pointer for interrupt processing. Finally, the interrupt base register contains the base address of the interrupt vector table, allowing this table to reside anywhere in memory.

### 2.2.2. Addressing Modes

Various computer architecture groups have referred to addressing modes as a subclass of instructions themselves. Viewed this way, the 16032's instruction format is:

Op      SubOp1, SubOp2

In referring to C on the 16032, the operators matched in C by 16032 addressing modes are essentially the structure, index, and dereferencing operators ( $\rightarrow$ ,  $*$ ,  $[]$ ).

The 16032 supplies 32 distinct addressing modes, most of which are modeled for use in high level languages like C. The following C program illustrates the addressing modes that are directly applicable to C and UNIX. Every subroutine line generates one addressing mode.



```

static int foo;
static struct two_words { int one, two; } *twop;
extern int bar;

main () {
    int local; struct two_words *local_twop;
    register reg; register struct two_words *reg_twop

    reg = 0;           /* direct register addressing      */
    reg_twop->two = 0;  /* offset register addressing */
    local = 0;         /* frame pointer relative    */
    local_twop->two = 0; /* frame pointer memory relative */
    foo = 0;           /* static memory relative    */
    twop->two = 0;      /* static memory relative    */
    bar = 0;           /* external address mode     */

    *((int *) 1234) = 0; /* absolute address mode    */
}

```

The basic addressing modes may each be augmented with indexing suffixes. Here is another brief illustration, which is the same as the previous one, except that each addressing mode is now indexed. As before, each line generates a single addressing mode.

```

static int foo[10];
static struct two_words { int one[10], two[10]; } *twop;
extern int bar[10];

main () {
    int local[10]; struct two_words *local_twop;
    register reg; register struct two_words *reg_twop;

    reg_twop->two[reg] = 0; /* offset register addressing */
    local[reg] = 0;        /* frame pointer relative     */
    local_twop->two[reg] = 0; /* frame pointer memory relative */
    foo[reg] = 0;          /* static memory relative     */
    twop->two[reg] = 0;     /* static memory relative     */
    bar[reg] = 0;          /* external address mode      */

    *((int *) 1234 + reg) = 0; /* absolute address mode    */
}

```

Finally, there is a Top of Stack addressing mode that the compiler uses to push and pop arguments on and off the stack.

### 2.2.3. Instruction Set Summary

The instruction set consists of the usual set of arithmetic operations plus a variety of special instructions. The instructions are all symmetric in addressing modes, as well as symmetric to byte, short, and long data types. The compiler needs no subroutines to simulate missing instructions for any data type. There is a calculate address instruction that allows rapid shorthand expression calculation by addressing modes. Single instructions implement all of C's arithmetic operators (+ - \* & | % ! ^ ++ -- ~). Boolean assignment instructions are available

for C's relational operators (`==` `!=` `<` `<=` `>` `>=`). There are bitfield instructions (extract and insert) that support the C bitfield data type. Finally, there are extensive string and block move instructions to support the standard C library functions. As a result, these functions (*strlen*, etc.) are implemented in as few as four instructions.

#### 2.2.4. Instruction Execution Speeds

Unfortunately, one can generate literally telephone directory sized listings of timing charts for this processor, due to the large number of instructions, addressing modes, and data types. Just with the *mov* instruction alone, there are  $3 \times 32 \times 32$ , or 3072 possible combinations. Here are a few key timings, for register to register, external to register, and static data area to static data area addressing mode combinations. Instructions appear along the left and their operands along the top. The table body gives the number of cycles the instruction takes.

|             | reg,reg | ext,reg | x(sb),y(sb) |
|-------------|---------|---------|-------------|
| <i>movd</i> | 3       | 35      | 25          |
| <i>addd</i> | 4       | 36      | 34          |
| <i>muld</i> | 85      | 114     | 110         |
| <i>divd</i> | 138     | 167     | 163         |

#### 2.2.5. Modular Software

A unique feature of the 16000 architecture is hardware support for modular programs. Originally designed to allow dynamic linking of ROM-based software libraries, this feature serves two functions: it isolates all absolute program and data references into special tables (to ease automatic relocation), and it reduces the size of code per module by reducing the number of full 32-bit offsets used in instructions.

##### 2.2.5.1. Concept of a Module

From a programming viewpoint, a module is a coherent expression of some self-contained programming abstraction, including data, procedures to manipulate that data, and references to entities defined by other modules. From the processor's viewpoint, a module is an object consisting of three parts: a code section containing machine instructions, a static data area containing data defined by the module, and a link table containing information describing external references made in the code section. A module has one additional, implicit, component: local data, consisting of arguments passed from a calling procedure and local variables of a procedure activation. The program counter, static base, and frame pointer registers mentioned above are associated respectively with the current module's code section, static data area, and local data area. There are addressing modes for each of these registers that allow references to each part of a module to be encoded in a very compact form. (This is true also of external references; they are described below.) Rather than a full 32-bit address, a given reference becomes a small, usually single-byte, offset from the register associated with the module component containing the reference. This scheme allows a significant increase in code density and reference locality at the cost of increased complexity when execution moves from one module to another; the new module's context must be established. This work is done automatically by the *cxp* (call external procedure) and *rxp* (return from external procedure) and related instructions. Absolute addresses still remain, but they play a much less significant role, describing the locations of things like i/o devices.

---

#### 2.2.5.2. The Link and Module Tables

The 16000 architecture regards a program as a collection of modules described by a module table. At any instant a program executes within a particular module; the module register contains the address of that module's entry in the module table. This entry contains three fields corresponding to the module components described above: the program (code) base address, the static data area base address, and the link table base address. When execution enters a new module, its module table entry is used to establish the new execution context.

A link table is a collection of entries, each describing an external reference made in the link table's module. There are two kinds of entries: external variable descriptors and external procedure descriptors. An external variable descriptor consists of the variable's absolute address. A procedure descriptor is a pair consisting of the address of the module table entry for the module containing the procedure, and an offset from the start of that module's code section.

#### 2.2.5.3. External References

There are two kinds of external reference, matching the two kinds of link table entries. An external variable reference names a variable defined in another module and is encoded using the external addressing mode. To access an external variable, the processor takes the following steps. First, it uses the module register to obtain the address of the link table. It then selects an entry from the link table by indexing the table with a displacement encoded in the addressing mode. Finally, it obtains the variable's address by adding a second displacement from the addressing mode to the contents of the link table entry.

The second kind of external reference is an external procedure descriptor. To transfer to such a procedure, the processor obtains a link table entry using the same steps it uses for an external variable reference. It then uses the first part of the link table entry to get the module table entry for the called module and uses this information to establish a new execution context. Finally, it transfers control to the new module by jumping to the offset in the new module's code section given by the offset field of the link table entry.

As can be seen from these descriptions, external references are expensive in terms of processor cycles consumed for memory references. There are strategies for speeding up such references. However, they are outside the scope of this paper.

#### 2.2.5.4. Software Requirements for Modular Software

Modular software requires the absence of any absolute addresses within a module either to itself or to other cooperating modules. Thus code must be generated entirely with relative addresses.

This causes some problems in C, where data objects may be initialized with addresses that cannot be resolved by the 16000 modular software until runtime. To avoid this problem yet retain modular software integrity, the compiler generates initialization code for run time addresses, and provides for a special entry point to execute this code at run time start off. Maintenance of such entry points is provided by the linkage editor.

---

### **2.3. The NS16082 Memory Management Unit**

The 16082 Memory Management Unit (MMU) works in concert with the CPU chip to provide a dynamic memory address translation environment. When installed, this device causes the CPU to reconfigure its operation to provide additional timing states as needed by the MMU to perform the address translation algorithm.

#### **2.3.1. MMU Specifications**

The MMU maps 512 byte pages to arbitrary page frames, thus allowing general demand paging. This means that the MMU accepts a 24-bit virtual address and provides (after translation) a 24-bit physical address. Only the high order 15 bits are translated, as the lower 9 bits which describe the address within a page need not be altered. Since there are over 32000 possible translations all of which could not fit on board the chip, the translation map (page table) resides in ordinary processor memory. To speed address translation, a cache of the last 32 different page translations is kept in a look aside buffer on chip. Cache update and write back is handled automatically and without processor intervention.

#### **2.3.2. MMU Architecture**

Page tables for a full size process occupy 129KB, a substantial fraction of the system's physical memory. At any time, much of this page table information is inactive, so it should be paged. The MMU has been designed with a two level paging structure to make it possible to page page tables themselves.

At the first level of mapping, the address space is partitioned into 64KB chunks; i.e., each first level page table entry governs access to a 64KB piece of the overall address space. The second mapping level further subdivides these 64KB pieces into 512 byte pages, with each second level page table entry controlling one page. Both first and second level page table entries contain access and protection information. It is important to note that first level page table entries stand in the same relation to second level page tables as second level page table entries do to data pages. This relationship is the key to why second level page tables themselves can be paged, and in fact can be managed with the same code that manages data pages.

Besides user programs, operating systems also require memory management and can frequently take advantage of virtual memory. For this reason the MMU supports two address spaces, one for supervisor mode and the other for user mode. Both address spaces are identical in function and neither requires the existence of the other space to function. Thus it is possible to develop virtual machines that may transparently function in either space. An important consequence of this functional equivalence is that space management mechanisms for both spaces can be the same, making it more attractive to page supervisor address space.

#### **2.3.3. Address Translation Algorithm**

In this implementation of the 16000 architecture, virtual addresses are 24 bits wide. To translate a virtual address into a physical address the MMU conceptually splits it into three pieces. The low order 9 bits determine an offset within page and remain unchanged in the translation. The high order 8 bits form an index into a level one page table and the middle 7 bits index a level two page table. A page table entry (level one and level two formats are identical) contains a page number field and protection information.

---

The translation process itself is as follows. First, the MMU gets the address of the level one page table from one of the page table base registers (which one is used depends on what address space is active). It then uses the level one index to select an entry from this table. The page number field of this entry determines the address of a level two page table. Provided there are no protection violations, the MMU continues by using the level two index to select an entry from this level two table. Again, provided there are no protection violations, the MMU finishes by concatenating the page number contained in this entry with the offset field from the original virtual address, yielding the corresponding physical address.

## **2.4. The NS16081 Floating Point Unit**

The hardware floating point unit was not available in time for our port of UNIX to the 16032, but was allowed for in our hardware design. (Our CPU board contains a socket for the chip; no hardware changes are required to incorporate it.) For completeness, here is a brief description.

### **2.4.1. Data Types and Format**

The floating point unit supports both single precision (32-bit) and double precision (64-bit) floating point. The data format conforms to the IEEE guidelines on floating point.

### **2.4.2. Instruction Format**

The instruction format is identical to that of the integer mode 16032 instruction set. Thus it is possible to have all combinations of addressing modes in floating point instructions, including memory to memory. This greatly simplifies code generation and optimization by retaining a regular model for instruction formation. The floating point instructions provide add, subtract, multiply, divide, and data conversion operations.

## **3. Differences in System Strategy from 4.1bsd**

Unlike 4.1bsd, we ported UNIX initially to support workstations and not timesharing systems. Thus we had some differences in overall system strategy from the Berkeley authors. However, some changes were necessary to support the new architecture, some were useful to bring up the system more rapidly, and some were things that anticipated Berkeley's new 4.2 UNIX system. Most of these changes have limited external visibility.

### **3.1. Debugger (*Ddt*)**

The main reason we were able to bring up the system as quickly as we did was the availability of a special debugger called *ddt*. This debugger, a distant functional relative of PDP-10 DDT, allowed us to breakpoint, single step, and stack backtrace the system (even at interrupt level). In addition, *ddt* is fully symbolic and can simultaneously work in both user and supervisor modes. The latter feature is great for debugging inter-address space problems, like following a system call from a user program down into the system and back out again.

*Ddt* can run both as an ordinary debugger (as a user process) and as a remote debugger. As a remote debugger, *ddt* runs on a separate machine, communicating over an RS-232 link to a tiny ROM monitor in the target machine. This arrangement is preferable to loading the debugger into the supervisor of the target machine, because it eliminates the inevitable side effects caused by the debugger's presence and by the need to specially configure the software of the system for debugging. It is simpler as well.

---

There are also MMU chip features that *ddt* supports as both a remote and a standard debugger. These features are unique to the 16000 chip set and are described below.

### 3.1.1. MMU Breakpoints

The MMU has two memory reference breakpoint registers that are accessible from *ddt*. These registers allow a user to break on reference to a portion of his address space. Thus it is possible to have the processor stop when a variable is changed. This is fantastic because one can find exactly where a complex data structure is being improperly modified! These data-space breakpoints are set by the user with the debugger just like any other breakpoint.

### 3.1.2. Program Flow Tracing

Another MMU debugging feature is program flow tracing. The MMU can break on nonsequential instruction execution, remember the last two nonsequential instruction branches, and remember the last two values of the program counter. The first feature supports a debugger that is sensitive to program control flow. The other two features are useful for revealing a program's execution history. Thus, a user can see a small snapshot of program activity prior to an exception such as a segmentation violation. The remote version of the debugger uses flow tracing when the processor has somehow gotten wedged and must be reset. The flow registers are not cleared on reset, so the debugger can show the last two addresses executed. This has been a great help in tracking seemingly spontaneous system failures!

*Ddt* supports the above features in both its remote and local forms. It also has many other features, more than space permits us to discuss. A future paper may illuminate the subject further.

## 3.2. New *a.out* Format

The 16000's modular software features forced us to modify the *a.out* file format. In the header we added a field specifying the module number of the entry point in addition to the field giving the entry point address; this information is necessary to establish the initial execution context. We added other fields giving sizes of other parts of the *a.out* file. The 4.1 format keeps the string table's size at the beginning of the string table; our format keeps it in the header. We also included fields giving the sizes of the module and link tables within the text segment.

The biggest difference in *a.out* format from the 4.1bsd version is in the symbol table; our format bears no resemblance to the 4.1 format. We dropped the old format because it did not mesh well with the modular software features of the 16000 architecture. Programs on our system are organized into modules, so each symbol mentioned in a symbol table is defined with respect to its containing module; thus our symbol table format defines entries that describe modules. Recall that addressing modes are intimately connected to execution contexts defined by modules. This relationship makes it convenient to define symbols for procedures and variables as the addressing modes used to reference them within a program.

Our symbol table format does not directly record address information for a symbol; instead, this information is carried implicitly through a combination of module entries, link table information, and symbol entries. However, many programs that extract information from *a.out* files neither need nor desire the wealth of information our symbol table format provides. These programs are exactly the ones that use *nlist* to obtain the desired information. Our version of

---

*nlist* caters to these programs by presenting the 4.1 interface to them; it translates from our symbol table format to the old one.

### 3.3. New Relocating Loader

Our modular architecture supports linking of a different kind than the standard UNIX loader *ld* provides. Relocation is provided entirely by the modular software features. Since relocation information is encoded in the executable form, a program can always be relinked. Program loading is more efficient than with *ld*, since only module and link tables need be computed instead of adjusting all of the addresses in all instructions.

Note also that the loader is able to link to ROM modules on the machine level.

### 3.4. Dynamic Page Table Management

Page tables themselves can be paged. In fact, page tables and ordinary data pages are handled by the same code in our kernel; only the bookkeeping structures are different. Usually a process only has a few active page table pages resident in memory; it faults in newly accessed page tables on demand with little overhead. An item that usually surprises people not very familiar with virtual memory systems is that system calls are usually much more frequent than page faults. Dynamically growable processes actually make the system simpler internally, since the additional code involved in managing a pool of static resources (e.g., process page table space) is not needed.

### 3.5. *Vspy* and *Vlock* System Calls

Our kernel supports two new system calls for address space management. They are functionally similar to the *phys* and *lock* system calls of Version 7 UNIX, but are different enough in actual semantics to warrant different names. The *vspy* system call remaps a previously unmapped page in a user process's address space to a given physical address. Unlike *phys*, the particular physical page is exclusively mapped to a process. This call is most frequently used to map i/o devices such as bitmapped screens into a process's address space.

The *vlock* system call arranges for certain pages of a process's address space to remain memory-resident until further notice. This feature is useful for real time applications.

## 4. History of the Port

### 4.1. Compiler and Basic Tools

The C compiler, assembler, debugger, and loader were first available in March, 1982. By the beginning of April we had made our first internal release of software porting tools. The early software had various known deficiencies. The C compiler had a calling sequence mechanism that totally ignored the variable argument features of C; this was a constant source of headaches. One of the major differences between this software and our current versions is that, for expedience, the initial versions used an executable format left over from a previous 16000 software project. Early versions of *ddt*, the assembler, and the loader were available from the previous project. We reworked them as needed to suit our porting needs. From an in-house curiosity, *ddt* eventually evolved into a product.

---

## 4.2. First UNIX Implementation

During April we created a reduced form of UNIX that did not require a memory management unit. This was important because no fully functional MMU chips existed, and experience suggested that we already had our hands full enough with new hardware. The system ran one user process at a time relocated to a high address to avoid the kernel. Independent parallel efforts took place to create and test drivers for the disk and bitmapped display. We also created a filesaver to allow the standalone utilities to transfer disk images from the VAX. During this month every aspect of the project was in frenzied action, with compilers and hardware being fixed on a daily (and even hourly) basis. Within a few short weeks, the system progressed from being barely loadable to executing */etc/init*. Because of problems in securing interrupts, the system was brought up without them, using instead a complex polling scheme. It was slow, taking about 10 seconds to run a trivial program. However, that success pushed hardware and software work forward at an equally frantic rate for another two weeks. At the end of the month we had a system that was definitely recognizable as UNIX, albeit somewhat simpler than the VAX system we had set our sights on.

A key success of the first implementation was that it enabled us to separate our problems into different categories. Because the system was simple, we could run it without total dependence on the hardware. The hardware could be used and tested without fancy system features pushing it over the brink. Since we had created a complete system, we were able to check out all aspects of the software in reasonable depth and discover hidden faults in our original plan. We indeed found several faults, in both our software and in our plan: the inadequate executable file format and the variable argument call problems turned out to cause more severe problems than we had expected. This simple system turned out to be very successful as a shakedown cruise for all involved, and continually served as a reference system during the next phase of the port.

## 4.3. Adding MMU Support to UNIX

Having received and experimented with a few MMUs, we proceeded to add code to use them. At first we used them only for protection, with the physical to virtual translation essentially an identity map. For simplicity we left the kernel's address space untranslated. This turned out to be a wise decision, for it was easy to distinguish MMU malfunctions from software problems. As we gained experience, we exploited more and more of the MMU's functionality, until finally we had a kernel that ran multiple processes. To aid debugging we extended *ddt* to deal with multiple address spaces, a critical step in its evolution.

## 4.4. Virtual Memory System

We continued to bring up the system gradually by supporting scatter loading. At this point page tables and "u." areas were forced resident to aid hardware debugging. The next step was to handle virtual memory for user pages. As soon as we shook this software down, we freed page tables and user structures to page on demand. We chose this conservative approach so that we could isolate and understand our software/hardware problems without additional code to obscure them.



---

#### 4.5. UNIX as a Microcode Debugger

During the course of porting the UNIX system to the hardware we discovered a number of problems with the 16000 chip set. Many manufacturers have cited UNIX as being an excellent diagnostic for their hardware, but we were a little unprepared for the critical exercise that the port provided our hardware. We detected subtle race conditions in our MMU only when we attempted to run a multitasking system with frequent context switches. At one point we found that an instruction was doing a dummy read of address 0 because we were running user programs with relocated address space. (Similarly, many UNIX programs have this bug.)

Our experiences lead us to wonder about the existence of benign bugs in other popular microprocessors. We suggest to future microprocessor implementors that early porting work is a very good idea! There is no test as thorough as that of running a full operating system.

#### 4.6. Graphics and Timesharing Support

As our workstations had only a bitmapped display for interactive output, we soon realized that we needed to have some way to use the display as a terminal. We chose to incorporate ANSI terminal emulation into the console device driver at the point where the driver receives characters to be output to the display. Initially, the emulation was very skeletal, but it has grown to the point where we can now treat the display as an intelligent terminal providing enough escape sequences (e.g., insert and delete line and character) to support editors efficiently.

Our graphics hardware provides no raster-op or bit-blt operations, so we have also had to devote a lot of attention to implementing these operations in software and to honing them to be as efficient as possible. The results are mixed. Running at 4Mhz, we can drive the bitmapped display (treated as a terminal) at an effective 2400 baud rate; this is visually disappointing and slow in use, but is impressive considering that we achieve this rate with no hardware assistance.

Support for the timesharing version of our system mostly takes the form of a change in emphasis; the underlying kernel software is, of course, very similar in both versions. The two systems support a different set of devices, so we have had to develop drivers for the new ones. The most important of these are a new console driver using a single channel RS-232 port on the CPU board, and a driver for an eight port serial i/o board. However, the most difficult part of adapting to a timesharing environment has turned out to be tuning the system. Multi-user operation places far more stringent demands on the system than running single-user does and has exposed weaknesses in our earlier implementation. We have made but a bare beginning in identifying and correcting problem areas; indeed, doing so is the major task remaining in our porting effort.

### 5. Future System Extensions

#### 5.1. Additional UNIX Features

With the flexible architecture of the 16000 chip set we can support a variety of desirable operating system features absent from most versions of UNIX. These features include flexible kernel address space management, memory-mapped files, shared memory, and a copy-on-write implementation of *fork*. Many of these features will be a part of Berkeley's 4.2 UNIX distribution; the architecture of our system will make it easy to support them.

---

#### 5.1.1. Kernel Virtual Memory

Kernel virtual memory can be handled with the same mechanisms as user virtual memory, provided deadlocks can be avoided (e.g., paging out the page fault handling code). As a feasibility demonstration, we have produced a version of our kernel in which the code for handling */dev/mem* is pageable. The biggest problem we had in producing this kernel was handling the bookkeeping for which kernel pages could potentially be nonresident; we had to wire this information in manually. With a more automated scheme for partitioning the kernel into resident and pageable sections (both code and data) and for arranging that each section occupies a contiguous stretch of the kernel's address space, several possibilities for increased kernel functionality become attractive. Large libraries of system calls could be included in the kernel at little cost in additional physical memory. It would be possible to have system call compatibility with several different variants of UNIX (such as 4.1bsd and System III) on the same machine.

#### 5.1.2. Memory-Mapping

Our kernel's internal architecture makes it straightforward to add system calls for altering address space mappings. The *uspy* and *vlock* calls discussed above are extensions of this kind.

Another such extension is support for memory-mapped files. Here, a file is bound to a region of a user process's address space so that memory reads (and possibly writes) within the region reflect (and affect) the contents of the corresponding part of the file. As page faults are taken in that region, pages can be filled from the declared disk file upon demand. If the file was opened read-only, the affected pages will be set to read-only protection, with write attempts trapped to the user process as segmentation violations. If the file was opened read/write, pages may be modified and the disk file is updated when the physical page is reused, or when the process terminates or otherwise forces a complete update.

Another closely related extension is inter-process shared memory. One way to view this feature is as an extension of the file memory-mapping facilities that allows multiple processes to map the same file into their address spaces simultaneously. This mechanism provides a way to set up very tightly coupled cooperating processes, with high-bandwidth communication through shared memory pages.

#### 5.1.3. Copy-on-Write

The 16000 architecture allows copy-on-write, a technique for reducing the cost of maintaining distinct, but closely related, virtual address spaces, such as those resulting from the *fork* system call. Copy-on-write works by initially mapping both address spaces into the same physical memory. This mapping is read-only, even for pages that are conceptually writable. Whenever one of the processes tries to write a shared page, the write aborts from a protection violation. At this point the kernel can change the address mappings to duplicate the offending page, giving each process a private, writable copy, and restart the aborted operation. Using this technique eliminates the need for the *vfork* system call, which uses a different, nontransparent strategy to reduce overhead from the *(v)fork* of a *fork/exec* pair.

---

## 5.2. Component UNIX

The 16000 family's modular software features open the possibility of a "silicon software" version of UNIX, in which the kernel text segment is ROM-resident. The most interesting issue to be resolved here is system configuration; in designing a ROM-based system, we do not wish to sacrifice the ability to generate kernels for varying hardware configurations. We envision a system in which the main body of the kernel and each device driver all reside on separate ROMs, with a particular configuration built by mixing and matching.

The 16000 architecture makes ROM kernels practical because it is possible to "patch" out modules by changing module table entries during startup configuration. Thus huge ROMs can be revised by changing module linkages instead of being replaced at great cost.

## 5.3. Present and Future Processor Support

We intend to implement this UNIX system on the soon to be available NS32032 processor. The 32032 is software-compatible with the 16032 and works with the 16000 family chip set. However, it has a full 32-bit wide data bus, twice the size (and bandwidth) of the 16032.

In addition to the wide bus, the 32032 supports dual processor operation with integral on-chip support (bus arbitration). Thus we intend also to implement a dual-processor UNIX system along the lines of the Purdue dual-780 system. This system will function as a master-slave multiprocessor unit.

## 6. Conclusion

We have described the history and status of our port of UNIX to the National Semiconductor 16000 microprocessor chip set. Further enquiries should be directed to National's 16000 marketing group.

### 6.1. Acknowledgements

We would like to acknowledge the work of the members of the Mesa project, both past and present. We especially wish to note David Bell for his enormous contributions to the design and implementation of the kernel. Finally, we thank Ross Harvey and Duncan Gurley for creating *ddl*. It was handy on many a bleak night.

---

# UNIX for the Computer Automation 4/95

*Steve Pozgaj*

Human Computing Resources Corporation  
10 St. Mary Street  
Toronto, Ont. M4Y 1P9 Canada

This talk discusses an implementation of UNIX on the Computer Automation 4/95 minicomputer. The 4/95 is a high-performance 16 bit computer featuring a modern memory management unit and a baroque instruction set. The major problems with this implementation involved 1) building a C compiler which handled word addressing properly, and 2) dealing with the fact that many of the machine features were not documented. This talk will be of special interest to those about to embark upon a UNIX port to any less well-known machine architecture. This talk will give tips on avoiding the many pitfalls in such a task.

## Architectural Implications of UNIX (or Pitfalls for UNIX Porters!)

*Matt Dickey, Greg Noel, Bob Querido et. al.*

NCR Corporation, SE-TP  
11010 Torreyana Rd.  
San Diego, CA 92121

*Bill Appelbe and Jim McGinness*

University of California at San Diego  
La Jolla, CA 92093

UNIX is being ported to an increasing variety of computer systems. The difficulty of porting an efficient UNIX implementation is highly dependent upon detailed characteristics of the target architecture.

The presentation discusses the constraints upon a *UNIX-compatible* architecture, as a guideline to new porting projects. The major architectural impact of UNIX upon memory management, address modes, primitive data types, operators, protection, and context switching is discussed.

UNIX assumes a low-level architecture (not necessarily a PDP-11!). Hence, it is usually easier to port UNIX to a simple architecture than to a high-level architecture which provides descriptors, and high-level task and memory management.

Chairperson: *Marlene Martin*  
The Marketing Network

## Getting Venture Capital

*Henry Wilder*

Dougery, Jones and Wilder Venture Capital

This talk was a concise summary of what it takes to get venture capital. The comments were also an excellent guideline for new entrepreneurs for choosing markets, products and business partners.

The steps followed by a VC (Venture Capitalist) in considering a proposal are the “path of greatest resistance” that eliminates unacceptable proposals as soon as possible. 5% of the proposals fail on introduction, 45% more after the VC has spent about 20 minutes considering details of the plan. After an initial presentation only 25% remain, and 10% more are eliminated during in-depth interviews. Another 10% fail when the VC and entrepreneur can not reach a mutually acceptable deal. The VC will then eliminate 3% more by “due diligence” — researching why the deal should be turned down business-wise and legally, leaving only 2% of the initial proposals that result in any investment.

The highest priority is that the proposal offer an inviting growth market with new customers available, little resistance to new entrants, positive economics (enough gross margin to support the business, distributors and dealers), and the possibility to become a large independent company. There should be evident ways to increase sales efficiency, and distribution channels should be available. The market should allow for products to be focused so that the company can pick the best products and sell in volume.

The third priority is a proprietary product plan. There should be something special and compelling about the products (capability, price, geography, reliability, etc.). There should be a barrier to competition (a learning curve head start, a lock on customers and suppliers, name recognition, etc.). The plan must offer a reasonable longevity to provide cash for establishing distribution and developing the next product. Finally, there must be some related future growth area for the new business to move to when the initial plan is accomplished.

The final priority of the VC that was discussed is to develop a reasonable deal. This includes having a step-wise investment schedule based on mile-stones to limit risk and having a reasonable cost for the profit potential. It is typical that terms for vesting the management in the resulting firm be based on seniority. The VC will be concerned about who else is investing and how much help and influence will be expected from the

investor group. It is important to keep the styles of all investors in the business consistent.

## UNIX Markets and Competition

*Bob Katsive*

Gnostic Concepts

Gnostic Concepts is a “professional reconnaissance unit” and has developed a strategic view of UNIX. UNIX is the result of a “demand/creation critical mass phenomena.” The university environment produced people who wanted a transportable operating system for inexpensive computers. There was a need for a standard interface that would be stable for third party vendors. AT&T supported this concept, and UNIX was born. Mr. Katsive declared, “If UNIX had not existed, it would have been necessary for man to invent it.”

---

UNIX is a fourth generation operating system with key features that reduce development times and costs. Hardware is becoming an inexpensive commodity, and inexpensive software is on its way. Some of what is coming includes invisible UNIX, silicon UNIX, distributed UNIX, very fast UNIX and secure UNIX. Mr. Katsive pointed out that the world's largest processing system is composed of all the UNIX sites on the ARPAnet.

UNIX-related EDP expenditures grew from \$151m in 1980 to \$1.38b in 1982. It is projected to be \$5.28b in 1985 with about 23% for support, 37% for personnel, 36% for hardware and 4% for systems. UNIX will be the fastest growing operating system on personal computers.

The greatest pressure on the future development of UNIX will come from the users of microcomputer hardware. In 1982 65% of the UNIX systems shipped were installed on microcomputers, 34% on minicomputers and 1% on mainframes. By 1990, 94% will be on micros, 3% on minis and 3% on mainframes. These estimates apply to the combination of Bell UNIX and UNIX look-alikes.

---

**This page intentionally left blank**

---

## Delivering UNIX to the End-User Market

*Michael Denny*

BASIS, Inc.

This was a delightful sermonette you had to hear in person to fully appreciate. In the words of the preacher, "I have been to the promised land, and it is a wilderness... What does it take to bring UNIX applications to the user market? Not Bill Joy."

Personal computer software is producing a groundswell of frustration. The end-user may be naive, but she/he is not dumb. CP/M is being pushed by the "media," but it is not the right answer. UNIX offers the real key, but care must be taken to meet the needs of the end-user.

No software is friendly enough. Users try to do their jobs, and the computer is supposed to help. You cannot expect the bulk of the users to learn operating systems — they need turnkey systems. With the computing power now available, large and expensive systems are unnecessary. There is no need to integrate the computer system and the end-user. You can broker application services to them so they can get on with their jobs without worrying with the operating system.

To have a good menu for an application, it is necessary to implement the user jargon for the job. Keep the menu from being too dry. Maybe conversational computing is better. And don't try to design the "perfect generic vocabulary." Make the software user-ready. It should be living software that is easy to reimplement and revise.

UNIX provides an unusually appropriate place for developing user-friendly applications programs. Mr. Denny concluded, "[Using this method means] the user can do his job, and we can do ours."



---

*SUMMARY*

**DELIVERING UNIX TO THE END-USER MARKET**

Wm. Michael Denney

It takes more than "friendly" software to bring computer applications to the end-user market. This conclusion comes from unique experience in managing a commercial Unix time-sharing service, plus professional involvement in personal computer market research. The concept of "user ready" is proposed to describe the gap between versatile software and productive end-user applications. To fill this gap, it is argued that a new kind of end-user consulting is needed — what might be described as "expert hand-holding."

The discussion focuses on barriers to self-computerization and shortcomings of conventional approaches to user-friendliness. To serve everyone generally is to serve no one well in particular. Although modern application software, such as database management systems and spreadsheet programs, offer both ease of use and an impressive array of features, the average business manager cannot afford the learning costs (especially time) required to implement useful, routinized computerization.

Custom implementation support is suggested as a promising strategy. Such support requires organizational diagnostic skills that are quite distinct from traditional computer expertise. Fortunately, Unix has special virtues in facilitating end-user consulting that is both effective and affordable. Software and hardware houses alike should welcome such a service.

---

# Distribution and Differentiation

*Marlene Martin*

The Marketing Network

This was a discussion of product and marketing considerations for selling in the UNIX market. The UNIX market is already a mature market with many kinds of applications available under UNIX. There is a shakeout coming in the market, and it is important for any young company coming out with a new UNIX-based product to be unique.

Differentiation of your product from others is important. One method of differentiation is the amount of value-added given to the product. UNIX plus a few tools may be sufficient for a barebones offering to an OEM. More tools may need to be added to produce a basic end-user product. Adding still more tools and features increases the product's value as an enhanced end-user product. You must choose just where you want to be in this differentiation.

Packaging of the product is vital. You must make your product look good to your customer. The amount of resources required to successfully package a product for the market will increase as its value-added increases and as you try to sell to the higher level markets.

Delivery is a commitment that must be met. Startup companies sometimes mistakenly see their goal as making money. They should see their job as offering a service that provides value to the customer. Timely delivery is one such service.

Distribution of UNIX-based products can be done through bundling with hardware systems, OEMs, value-added resellers, master distributors, industrial distributors, retailers or direct to the end-user. UNIX products have not been an easy sell in the retail markets because of the difficulty of training sales personnel.

Some suggestions for OEM marketing are to emphasize reliability, delivery and features. Do not compete with your OEMs and avoid dealer over-crowding. In conclusion Ms Martin said, "Put your marketing strategy in place and stick to it."

## New UNIX Markets in Engineering

*Camran Elahian*

Computer-Aided Engineering

The right application software using the features of UNIX represent a winning combination in the CAD market. This talk gave an example of the application of UNIX to a particular vertical market.

CAD was spawned by the breakthrough of minicomputers and inexpensive graphics hardware. The market will grow to \$4b in 1986. First generation CAD systems have about 80% of the current market, but their lack of standard components has minimized portability, diluted efforts and lengthened product cycles.

At least 27 second generation CAD systems have emerged in the early 1980s as a result of the introduction of microcomputers, better graphics and Local Area Networks. Some standard components have emerged, but portability remains a problem. By using UNIX as the base, the speaker's company has developed a portable CAD system featuring a common data base for CAD as well as CAE and CAM. The product is a good example of how UNIX can be the basis for vertical-market systems that are portable and that make efficient use of design efforts.

---

Chairperson: *Mike O'Dell*  
Lawrence Berkeley Laboratory

## **Portability in the UNIX World**

### **What UNIX Can Learn from the Software Tools**

*Mike O'Dell*

CSAM 50B/3238  
Lawrence Berkeley Laboratory  
University of California  
Berkeley, CA 94720

UNIX is a living, breathing, growing system which has become famous for its ability to change and adapt. Now people are beginning to fret because this growth and change inevitably leads to non-portability of programs, even within the UNIX world. Luckily, the Software Tools provide a well-developed and tested technology for dealing with diversity of environments. The Tools technology is built on the shoulders of the ideas central to "UNIX-ness," but in a highly portable way. This technology will be explored in some detail and its implications for portable UNIX programming will be examined. This confluence is particularly fortunate because of a growing interest within the Software Tools Group to adopt C as the successor to Ratfor.

If these ideas do not prove sufficiently controversial, the speaker will advocate adoption of EBCDIC as the standard character code for all programming.

---

# A Tutorial on C Portability

*Michael Tilson*

Human Computing Resources Corporation  
10 St. Mary Street  
Toronto, Ont. M4Y 1P9 Canada

C programmers have always been aware that it is possible to write programs in C which depend on the underlying hardware. The C language allows portable programming, but does not require it. A number of people have experience with 11-to-VAX-to-68000-to-Z8000 portability. Indeed, there are a number of serious issues in moving between these machines, many of which have been discussed in talks and articles by other authors. Recently, the UNIX system has been moved to a number of machines which are less "friendly" to C. This talk is a tutorial on C portability, illustrated by examples. It will discuss issues of program portability and style which are important to anybody producing widely portable UNIX software. Most C programmers will not have been aware of all of the specific issues discussed. The talk draws upon the author's experience with the PDP-11, VAX, Z8000, 68000, CA 4/95 (a word-addressed machine), Three Rivers PERQ (a word-addressed stack architecture), and NS16032. The talk will contain some advice for compiler writers, and will also discuss some flaws in the C type rules.

---

A TUTORIAL ON C  
PROGRAM PORTABILITY

Michael D. Tilson  
Human Computing Resources Corp.  
10 St. Mary Street  
Toronto, Canada M4Y 1P9

416-922-1937

decvax!hcr!mike

ABSTRACT

This paper is a short summary of a tutorial talk given at the January 83 Unicom meeting in San Diego. This is not a full paper, but rather an extended abstract of the talk.

## THE PROBLEM

Imagine that you have just received a big program from your friend at X University. You go to read the tape and compile the program:

```
$ tar x bigprogram.c
$ cc bigprogram.c
$ a.out
Segmentation violation - core dumped
```

This isn't the desired result!

## C LANGUAGE PORTABILITY

This talk focuses on machine-independent coding in the C programming language. The talk is illustrated by examples. All of the examples have arisen in practice. This talk does not cover a number of related issues, such as portable library routines, operating system interface, or the command interface.

Why do we want portability? Computers are becoming a commodity. When you buy a computer, you want to ask a few questions: Does it run UNIX? How fast is it? How reliable is it? How big a program can you run? How much does it cost? You don't really want to ask very many other questions. In particular, you don't want to be forced to continue to buy from the same vendor if another computer is faster and cheaper. We want to reuse our software without any conversion cost.

I might also add that portable software tends to be more reliable, since it tends to be less "dirty", and use fewer "tricks". Errors in type matching often indicate errors in thinking.

HCR has a strong commercial interest in portability. We supported or developed UNIX versions for the PDP-11, the VAX, the VAX under VMS, the Three Rivers PERQ workstation, the Computer Automation 4/95, and the NS 16032 microprocessor. We have also used UNIX on several other processors, such as the 8086 and 68000. We sell UNIX software products on these and other machines. We estimate that non-portable software has cost us well over \$100,000 in the last year.

## LEVELS OF PORTABILITY

There are two possible kinds of portability. The first level of portability produces programs which are completely machine independent, completely type safe and size safe. The second kind of portability involves portability of resulting binary files

---

from one "reasonable" machine to another, for example between machines which have 8 bit chars, 16 bit shorts, and 32 bit longs. For example, you might have a C cross compiler for the CA 4/95 which initially ran on a PDP-11, got moved to a VAX, and then got moved to the 4/95 itself. This kind of program is not theoretically fully portable, but can still be made easy to move over an important class of machines.

## PORTABILITY PROBLEMS IN C

C is not a safe language. The C "white book" describes a number of portability problems. To avoid portability problems, you must be familiar with all of the footnotes. Problems commonly arise from the careless use of pointers and the mismatching of types, from assumptions about the sizes (in bits) of various types, and from alignment assumptions (such as the order of chars within a short int.) Type casts and lint don't solve all problems. Type casting can make lint shut up without solving the portability problem. There is no substitute for getting it right. Ric Holt of the University of Toronto describes C as a "high level assembler". While he exaggerates, it is true than many C programmers visualize the resulting compiled code as they write C programs. This may result in greater efficiency on some machines, but it does not make for portable software.

## EXAMPLES

The remainder of the tutorial will focus on actual programming examples which have arisen in HCR's work. This is not an exhaustive list of what can go wrong, but rather an illustration of the care which you must take when writing portable programs. I should add that all of the examples are lintable, at least with the default level of lint checking. In the following examples, you will see calls to "abort". On the machines most people are used to, "abort" will not be called. However, for each example there are machines for which the "abort" call will happen.

### Example 1:

```
int *p;
char *q, *q2;
...
q2 = q;
p = (int *)q;
q = (char *)p;
if(q != q2)
    abort();
```

---

The above can fail because there is no guarantee that the conversion of a pointer from "char \*" to "int \*" will preserve all of the bits of precision if the "char \*" had not already been aligned to an "int" boundary. The above works on the VAX, PDP-11, and most other machines, but will fail on the Harris /6 and on some other machines. This kind of type mismatch is common in UNIX software.

Example 2:

```
int *p;
char *q;
int arr[50];

p = &arr[20];
q = (char *)arr;
q++;
if(p < (int *)q)
    abort();
```

How can this fail? Variable "p" is clearly at a higher memory address than "q". Not true. The "q++" could produce an internal bit representation which, if considered as an "int \*", would look like a word pointer at a larger memory address. This example is adapted from the Bourne shell, and fails on the PERQ and the CA 4/95.

Example 3:

```
int *a, *b;

a = (int *)sbrk(0);
b = a;
b--;
if(a < b)
    abort();
```

What could be wrong with this? On a segmented architecture, "sbrk" might return an address at the start of a memory segment. Pointer arithmetic is only valid if the pointers remain within a known contiguous and properly aligned storage structure. The "b--" could wrap around to the high end of a memory segment. This example comes from a version of the UNIX text editor.

Example 4:

```
execl("/bin/echo", "echo", "hello", 0 );
```



---

This error is from the UNIX Programmer's Manual. The constant "0" is guaranteed to be a valid null pointer if used in an expression. However, C does no type checking across function calls. On a machine with 32 bit pointers and 16 bit ints, the above could cause trouble (and does with the HCR PERQ C compiler and with at least one Bell Labs 68000 C compiler). It would be better to use:

```
execl("/bin/echo", "echo", "hello", (char *)0 );
      or
execl("/bin/echo", "echo", "hello", NULL );
```

where "NULL" might come from a standard include file (e.g. "stdio.h").

#### Example 5:

```
int x;

x = 70000;
if(x != 70000)
    abort();
```

This is simple. The above works on a VAX, but not on the PDP-11, because the number 70000 fits in an "int" on the VAX, but not on the 11. Moral: always explicitly specify "long" if a number is going to be larger than somewhat. (A reasonable "somewhat" is 32767.)

#### Example 6:

```
char *p, *q;
int n;

n = (int)p;
q = (char *)n;
if(p != q)
    abort();
```

The C language guarantees that there is a large enough int to hold the bit representation of a pointer. However, on a machine with 32 bit pointers and 16 bit ints, the above will fail because the conversion to int will overflow. Conversion to anything other than "long" is not portable. (By the way, conversion of pointers to integers is in general a bad idea, and is always non-portable if the value of the "int" is ever examined. The only safe thing to do is convert back to the same kind of pointer.)

Example 7:

[Assume this is running on a machine with a large (e.g. megabytes), contiguous, uniform address space.]

```
int *p, *q;
long int n;

n = 0;
q = p;
for(i=0; i<5; i++) {
    q += 32000;
    n += 32000;
}
if((q-p) != n)
    abort();
```

The subtraction of two pointers is defined by the manual to yield "int". Once again, int might not be large enough to hold the difference. This might be an unfortunate restriction in some C compiler, but it is dictated by the current C definition.

Example 8:

```
struct {
    short int a_magic;
    long int a_text;
    long int a_data;
} header;

(void) write(f, (char *)&header, sizeof header);
```

What could be wrong with that? Everything is nicely type cast, and even the return value from "write" is fastidiously voided. The problem is that a binary structure is being communicated to the outside world via the "write" call. This always indicates a possible portability problem, since the resulting file can't be moved from machine to machine. In the future, we may see networks of dissimilar UNIX machines, but with network-wide file systems. One should be careful about producing binary files, and one should never assume that they will be portable.

Example 9:

```
long int n;
printf("%d\n", n);
```

This example is the bane of everyone who has ever had to move

---

programs from the VAX to the PDP-11. The "%d" prints an "int", which on the VAX is the same as "long int", but not on the 11. This is an example of the more general case of function argument mismatch, but one which is not checked by "lint" (although it really should be.)

Example 10:

```
double x = 1.234e30;
printf("%d\n", x);
```

This is an example of a common programming error. This example was recently used in a very glossy advertising brochure sent out by a company promoting its C training courses. It looks like the programmer wanted to print the number as decimal digits truncated or rounded to an integer. The fix proposed by the glossy brochure is as follows:

```
printf("%d\n", (int)x);
```

On our VAX, the above "fix" prints "0", rather than the correct number. This is an example of throwing in a type cast to patch a problem, rather than getting it right. Here a type cast was used to try to correct an error. In other cases casts are used to remove portability problems. In all cases there is no substitute for simply making things match up correctly in the first place. Here is a much better solution:

```
printf("%.0f\n", x);
```

Example 11:

```
to = bp->b_ptr;
asm("movc3 r8,(r11),(r7)");
bp->b_ptr += put;
```

This bit of program was supplied by an institution which requires any use to be acknowledged, so from the file which contains this bit of filth I reproduce the first line:

```
/* Copyright (c) 1980 Regents of the University of California */
```

This program fragment illustrates the extreme case of non-portability. "Asm" is a keyword in the "portable" C compiler. It causes the string argument to be emitted into the assembly source which results from compiling the program. In this example, there is no comment to say what is going on. The programmer knows what C variables are in particular registers. (The variable "to" is one of the registers used in this VAX assembly instruction. You guess which.)

---

If you feel you must write such "efficient" code, at least write it a bit more cleanly:

```
#ifdef vax
    asm("movc3 r8,(r11),(r7)");
#else
    --- insert portable C equivalent here ---
#endif
```

Example 12:

```
int TheQuickBrownFox;
int TheQuickGreyFox;
```

This causes no end of trouble when moving code from Berkeley VAX systems to other systems. While meaningful variable names are laudable, names which are not unique in the first seven characters are not portable to other UNIX systems. With a little thought, you can always choose nice names which are also portable:

```
int TheBrownQuickFox;
int TheGreyQuickFox;
```

These examples are only an indication of the kinds of portability problems are often found in real UNIX programs. You can avoid problems by viewing the machine as an abstract entity, and not making assumptions which depend upon actual bit representations. With a little bit of care, your software will be usable on a wide range of machines. If your software is worth using more than once, then it should be portable, since it's worth using on more than one machine.

---

**This page intentionally left blank**

---

# IS/3: A Compatible Extension of UNIX System III

*Steven Zucker*

Interactive Systems Corporation  
1212 Seventh Street  
Santa Monica, CA 90401

Interactive System/Three (IS/3) is a family of UNIX-based systems for micro to main-frame computers. The members of the family are compatible extensions of UNIX System III. This presentation discusses the nature of this compatibility, its benefits, and its costs.

Some of the topics we discuss are:

- Why standardize?
- Why standardize on System III?
- Supersets versus "vanilla-flavor" implementations.
- Some hard lessons from the old days.
- Compatibility of the user and program interface.
- Programming standards and porting procedures.

We also discuss some of the more recent lessons we learned in the process of porting IS/3 to a variety of hardware environments.

---

## IS/3: A Compatible Extension of UNIX\* System III

Steve Zucker  
INTERACTIVE Systems Corporation  
1212 Seventh Street  
Santa Monica, CA 90401  
...!decvax!cca!ima!ism780!steve

INTERACTIVE System/Three (IS/3) is a family of UNIX-based systems for micro to mainframe computers. The members of the family are compatible extensions of UNIX System III from AT&T. This presentation discusses the nature of that compatibility, its benefits, and its costs in the context of hard lessons learned in the days before there was an accepted UNIX standard.

### Why standardize? Why standardize on System III?

End users and vendors of hardware want to be able to draw on the massive amount of software that has been and will be developed for UNIX. Vendors of application software will not want to restrict the market for their products to unique features of particular systems, even their own, and so will tend to code to the prevalent standard. In this context, any standard will serve so long as it is not excessively restrictive or hard to implement on most hardware.

System III and its successors as they come from Bell will set the standard for UNIX systems not only for historical reasons -- AT&T/Bell has always set the UNIX standard -- but also on their own merits. System III is not without warts, but all things considered, it is a remarkably clean system. Given the heavy production-oriented uses of UNIX within the Bell system, it is reasonable to expect that any future systems from AT&T will be evolutionary rather than revolutionary steps from earlier systems, so its probable stability provides another reason for its adoption.

While we all have wish-lists of features we would like to see, adherence to the standard will provide provide greater payoff in the large and growing UNIX marketplace than failure to standardize, and it is better to adopt an available and attractive candidate like System III (soon, System V), even as on an interim basis, than to delay until all parties can agree on a final standard.

---

\* UNIX is a trademark of Bell Laboratories.

---

### Some hard lessons from the old days

INTERACTIVE was the first commercial supplier of UNIX systems. It introduced its first product, IS/1, in 1977, well before UNIX Version 7 was available. Over the intervening years, INTERACTIVE has developed and marketed a number of packages for use with UNIX, including an editor and word processing system, an electronic mail system, networking products, C cross-compilers for various microprocessors, and others. These products were initially designed to work with IS/1, which evolved over the years to provide features that were not in the base system. Some of the products were made to work on other versions of UNIX as well, though sometimes substantial effort was required. The support and maintenance of those applications on non-IS/1 systems became a nightmare as the systems began to diverge. Over time, it came to be realized that adding features to the operating system, even though well-considered and motivated by the best of intentions, was providing marginal improvements to the applications on IS/1 at the expense of portability to the UNIX marketplace as a whole. Furthermore, programmers became reluctant to use the extended features because doing so led to maintenance problems.

In addition to exacerbating the "export" problem (exporting IS/1 applications to other UNIX systems), system extensions sometimes produced an "import" problem as well. As the system evolved, some extensions were introduced that made it less easy to run programs produced on other UNIX systems. In the historical context of the late 1970's, this was not unusual or avoidable because there was no real UNIX standard. Asking someone what system he was running would often produce an answer like "Version 6 with the additions from Rand on the user group tape and the fixes that fell off a truck at the Illinois meeting, plus a few local additions." Fortunately, this situation is changing for the better as the UNIX community as a whole recognizes the benefits of standardization.

### U&PI Compatibility

We have defined a compatibility standard for IS/3 called User and Programmer Interface (U&PI) compatibility which, though strict, does not rule out enhancements, even to the underlying operating system. Instead, it provides guidelines for determining what enhancements are acceptable from the point of view of application portability. U&PI compatibility with System III means first that programs and sub-routines that are available to ordinary users in System III will behave in the same way on IS/3 (modulo bug fixes). This solves the import problem while permitting compatible extensions to commands (addition of options) and improvements in performance through improved



---

algorithms. And of course it allows the addition of new commands and subroutines.

To mitigate the export problem we have ruled out most extensions to the operating system that cannot be emulated in a satisfactory manner with subroutines built on System III primitives. This does exact a cost by not allowing major functional improvements to the system but it is the only way to insure that our applications will run on non-IS/3 systems based on System III. Unless a software vendor is in a position to set the standard or is willing to restrict the marketplace for his applications, this is an unavoidable consequence of standardization.

### Examples

U&PI compatibility does permit more efficient or reliable implementations. For example, some members of the IS/3 family of systems use 1024-byte or larger disk allocation units, implemented as clusters of 512-byte blocks. When users specify, or programs report, block sizes, they do so in terms of the System III standard of 512-byte units; the clustering is treated as an implementation detail for efficiency and does not intrude on the user view of the system. Similarly, System III does not provide a secure way of determining the login name of a user, but provides a subroutine that does the best it can based on available information. We intend to add a system call to provide secure and more efficiently obtainable login names. Our applications will use the extension, but we will have guaranteed that they will run as well as possible on non-IS/3 systems.

We also intend to add intelligent terminal echoing (e.g., visual indications of interrupt, quit, and end-of-file) and terminal paging (auto-stop at the end of a "screenful"). The compatibility standard dictates that these options be invoked in a way that will not conflict with present or projected extensions of the System III terminal modes and that the options not be embedded in applications intended for export. There will be no documented program interface to the features to tempt the unwary; instead, there will be local commands to set the modes almost as if they were provided by the terminal itself rather than the system.

Our U&PI compatibility criterion does not extend to system management utilities and procedures. There are a number of reasons for exempting these facilities. They are not intended for export to other systems and are generally invisible to most users and applications programmers. They are also the weakest areas of System III, and many of them are system (hardware) dependent.

---

### Programming standards and porting procedures

The interface to system extensions has also received attention. Our earlier experience indicated that a sharp division between the base standard (System III) and our extensions was desirable. For example, instead of mingling our subroutines and include files with those of System III, we have separated them. This makes the uses of extensions obvious and prevents name conflicts with future AT&T systems or with files added by other system vendors. This has greatly reduced the time and effort involved in moving our applications to other systems.

For exporting applications to other systems, we first port IS/3 include files and a compatibility library that provides the best emulation of the extensions using primitives available on the target system. If the target system is faithful to the System III standard, this task is done once only. Since we use the standard libraries and include files on the target system, any local changes on the target system that are reflected in those libraries pose no problem, as they would if we used the versions on our own system.

Chairperson: *Joseph Yao*  
Science Applications Inc.

## VMS C Compiler

*Jean Wood*

Digital Equipment Corporation  
Technical Language Group  
110 Spitbrook Road  
M/S Zk02-3/N30  
Nashua, NH 03060

DEC's new VMS C compiler began with an existing PL/1 compiler which was restructured and implemented with a switch to generate C code when desired. Five languages (C, PL/1, Macro, Bliss and TBL) were used in coding the various modules. An LALR parser, interacting with a lexical analyzer, passes the processed input through a semantics routine which produces highly optimized native-mode VAX code.

The primary design criterion for the compiler was portability with UNIX. It was intended to promote usage of VMS on the VAX. However, some small portability problems still exist. Not all, but a large portion, of UNIX functions are supported.

The second version of this compiler will not have a C device-driver, but may have full debugging support. It will be packaged as a set of four rather than five disks, and hopefully will also be available on magnetic tape. No serious bench-mark studies have been done yet, but it was reported that several programs studied thus far have run at an improvement of around fifteen percent. Higher compilation rates were also in evidence.

## UNIX APL

*Joseph Yao*

Science Applications Inc.  
1710 Goodridge Drive  
McLean, VA 22102

APL is popular in scientific programming due to the power of its large operation set. Most of its operations can be performed on either arrays or atoms. One portability issue with APL is that most versions compile code for the character set of a specific terminal. APL's character set is very large, including underscored characters, so this is a great problem for programs that run on different terminals. In this IBM UNIX version this problem has been solved — the system will set the environment for a specific terminal when the user inputs a terminal-type code.

I/O with a file variable can be achieved using the shared-variable function, SVO, which is also implemented in this version.

The *yacc* parser, APL.g, and lexical analyzer, lex.c, parse the program text to strings which are later interpreted. Some difficulties existed in the areas of statement printing within functions and in file I/O, but, for the most part, they have been resolved.

---

# The NIAL Language Project

*M. A. Jenkins*

Queens University  
Department of Computing and Information Science  
Kingston, Ontario  
Canada K7L 3N6

Q'NIAL is a Queen's University implementation of Nested Interactive Array Language. The primary goal of the language is to match the data-structures and operations to the conceptual structures of the mind.

All data-structures are arrays with zero to n-dimensions. (A zero dimensional array is considered to be an atom.) Any arrangement of these arrays can be declared, and the frames within the arrays may be of different types, allowing the power of a record. The transformer, "EACH," can be used to apply one of a vast selection of functions to each array element.

The implementation of Q'NIAL is in C, aiding portability. It contains many of the list operations of LISP, as well as a powerful operation-set similar to that of APL. The VAX-UNIX version is in test-site use currently. Packaging and licensing arrangements are still being made, and the product is expected to be ready in late Spring.

---

## The Nial Language Project

M. A. Jenkins  
Computing & Information Science  
Queen's University

Nial is a new general-purpose programming language designed as a very high level tool. Q'Nial is a portable implementation of Nial being developed at Queen's University, Kingston, Canada. The VAX UNIX version of Q'Nial is nearing completion. It is in test site use at 10 institutions including MIT, U of Toronto, U of Alberta, and the Technical Universities of Norway and Denmark. The system is expected to be completed for delivery in the spring.

The Nested Interactive Array Language (Nial) has been designed jointly by Prof. Mike Jenkins of Queen's University and Dr. Trenchard More of IBM Cambridge Scientific Center. It is a unique language in two respects. First its data objects and the operations on them are governed by a mathematical theory of nested arrays. The theory has guided the development of the expression sublanguage within Nial. Second, it has been carefully designed to synthesize many current ideas in programming languages. It encompasses ideas from Lisp, APL, SETL, structured programming, and functional programming in a coherent framework.

The Q'nial interpreter is written in C. It is designed to run under a number of operating system environments. Although it has been developed in a Unix environment its interface to a host system is general enough to allow it to be implemented in other environments with little change. Earlier experimental versions have been ported to TOPS-20, PDP-11 Unix, Amdahl's UTS, and PC DOS. We expect to port Q'Nial in its final form to all the above systems plus a variety of Unix systems.

Queen's University intends to make Q'Nial available on a licensed basis. The details are still being worked out, but the intention is to have different licensing arrangements for various classes of users such as individual researchers, teaching institutions, commercial users, and system suppliers. Prices have yet to be determined but the intention is to make them attractive enough to spread the language while recovering sufficient funds to continue development of the system.

---

# SOLID: for On-Line Systems Development

*John R. Mashey*

Bell Laboratories  
WH 1B-221  
Whippany Road  
Whippany, NJ 07981

Solid/UNIX is an update of the 1982 version of Solid. Solid is portable, has a common information database, and is group oriented. The system interfaces individual modules perfectly, eliminating a need for customization between group programmers. It also provides an excellent environment for individuals.

A document universe is set-up by the system, and modularity is promoted by re-use of as much documentation as possible. Files under the system are protected from collisions (simultaneous update of a file). Mostly implemented in C-shell, the system is fairly mobile.

The primary storage mechanism for a file is *sccs* — the source code is never discarded but kept in a “dust bin.” In general, SOLID provides more structuring than *make* but *make* still has advantages for certain applications.

SOLID is currently in use by approximately thirty BTL projects of various sizes.

---

SOLID - System for On-Line Information Development

John R. Mashey  
Bell Laboratories  
Whippany Road  
Whippany, NJ 07981

ABSTRACT

SOLID is a portable, extensible toolbox and component organizer for managing all the pieces of a project and its development environment. It has evolved from many years work in building project support environments. It attempts to give a new project a reasonable default environment at project inception, but provide tools for customization to local needs. From our experience, most projects end up creating customized environments, but there exist many similarities among those environments. SOLID is (an apparently successful) attempt to build something to handle both the common functions and the customization that always seems necessary. As of 10/82, it is used by about 30 BTL projects for purposes ranging from management of large (>5000 pages) documentation efforts, to total project development for micro through maxi target processors.

Chairperson: *Heinz Lycklama*  
Interactive Systems Corporation

## The /usr/group Standards Committee

*Heinz Lycklama*  
Interactive Systems Corporation  
1212 7th Street  
Santa Monica, CA 90401

The /usr/group Standards Committee was formed in the summer of 1981 by the Board of /usr/group in response to the need for a standard in the commercial UNIX marketplace. The existence of a standard will benefit both producers and consumers of UNIX-based software. The committee has put great emphasis on establishing the proper organization for this effort so that the proper procedures are in place when the time comes to vote on a standard. The committee is sensitive to legal and competitive issues involved with attempting to establish a standard. Efforts are currently being focused on identifying the set of system calls and sub-routines which will comprise the system interface standard and on identifying a set of extensions to the system interface which are frequently required by commercial applications.

This paper will discuss the purpose, charter and procedures used by the /usr/group Standards Committee. It will discuss the current status of the standardization effort as well as the schedule for adopting a standard. Some details of the proposed standard will be described.

The draft standard may be purchased for \$50 from  
/usr/group  
P.O. Box 8570  
Stanford, CA 94305



---

/usr/group Standards Effort  
UNICOM  
January 28, 1983

Heinz Lycklama

## 1. Introduction

The /usr/group Standards Committee was formed to formulate, adopt, publish, and provide a formal standard specification, based on the UNIX\* operating system developed by Bell Laboratories, for a commercial operating system. The purpose, benefits, and format of the Standards document are discussed. This document is intended to give the reader an appreciation for the scope of this effort. Some of the history of the commercial UNIX system market is covered to give the reader a historical perspective.

The /usr/group Standards Committee was formed in the summer of 1981 by the Board of /usr/group in response to the need for a standard in the commercial UNIX marketplace. The existence of a standard will benefit both producers and consumers of UNIX-based software. The Committee has put great emphasis on establishing the proper organization for this effort so that the proper procedures are in place when the time comes to vote on a standard. The Committee is sensitive to legal and competitive issues involved with attempting to establish a standard. Efforts are currently being focused on identifying the set of system calls and subroutines which will comprise the system interface standard and on identifying a set of extensions to the system interface which are frequently required by commercial applications.

The benefits of having such a standard are enormous in the exploding UNIX applications marketplace. Interchange of programs, data and personnel is greatly facilitated. The educational and economic benefits are self-evident.

The committee is composed of about 30 key representatives from the UNIX-based system, UNIX-like system and UNIX applications vendors, as well as from the community of UNIX users. The recent addition of representatives from Bell Laboratories has strengthened the ability of the Committee to define a standard on which a large section of the UNIX community can agree.

## 2. Versions of UNIX

The UNIX system was developed at Bell Telephone Laboratories in the early 1970's to provide a convenient software development system. It was originally written for the PDP11 computer in assembly language. One of the most significant developments in this effort has been the fact that the complete system was rewritten in the low-level systems

---

\* UNIX is a trademark of Bell Laboratories.

---

implementation language, C, for the PDP-11/45 computer in 1973. In 1974, Version 5 of the UNIX system was released to educational institutions. It gained popularity very rapidly. This led to its availability in the commercial market in 1976. In 1977, a number of companies started to provide support for the UNIX system to commercial sites. The next version of the UNIX system, PWB UNIX, was released in 1977 as well. However, the high costs of source licenses and binary licenses prohibited the wide adoption of UNIX systems at commercial sites. In the meantime, the availability of UNIX licenses to educational institutions for a nominal distribution fee encouraged the widespread use of UNIX at universities. Later versions of the UNIX system (Version 7 and System III) had lower binary prices and less restrictive licensing, thus increasing the popularity of the UNIX system in the commercial world.

One of the main stimulants to the standardization efforts has been the fact that there are now a large number of versions of UNIX and UNIX-like systems being marketed commercially. This includes versions of UNIX as distributed by AT&T, UNIX-based systems marketed by some commercial vendors, licensed by AT&T, and UNIX-like systems marketed by companies without a license from AT&T. This poses a real problem for those who are attempting to develop applications for the UNIX marketplace. What systems interface do the applications vendors program to?

There are a number of versions of the UNIX operating system available from AT&T. These include V6, PWB, V7, 32V, System III and System V (expected in February 1983). The later versions of the UNIX operating system are distributed for both the VAX and the PDP11 computers. Internally, Bell has standardized on System III, and subsequently, on System V. The intent is to make later versions of the UNIX system available to commercial vendors at the same time as internally to Bell. AT&T has also made a commitment to make later versions of the UNIX system upward compatible with previous releases, starting with System III.

The majority of the UNIX systems vendors market UNIX-based operating systems. That is, they have modified and/or extended the original UNIX operating system, as released by AT&T and remarketed the system under their own names. The most recent systems are based on UNIX System III. One of the predominant exceptions to this is the Berkeley 4.1BSD system for the VAX computers, which is based on 32V. The common trade names used for the more popular UNIX-based operating systems include IS/3, XENIX, UNIPLUS, UNITY, VENIX, Zeus, ONIX, UTS, etc.

Then there are the UNIX-like operating systems, which the vendors claim are compatible with the UNIX system in anywhere from spirit to complete specifications and utilities. The IDRIS system, marketed by

---

Whitesmiths, claims to be largely V6 compatible. The UNOS system, marketed by Charles River Data Systems, and the Coherent system, marketed by Mark Williams, claim to be largely compatible with V7 UNIX, with some extensions of their own. Other UNIX-like systems have also been announced recently.

With this diverse set of operating systems, the need for standardization arises.

### 3. History

Early in 1981, the /usr/group Board of Directors determined that a formal standardization effort could be of significant benefit to the general membership of /usr/group. Heinz Lycklama was appointed to serve as Chairman of this standards effort and to organize a Committee that would propose, define and publicize this standard. The first organizational meeting was held in conjunction with the /usr/group meeting held in Los Angeles during July of 1981. About forty-five people expressed interest in working on the /usr/group Standards Committee. A sense of the charter of the committee was developed at this meeting. The intent was to provide a broad representation of systems vendors, applications builders, and end-users in the UNIX marketplace on the committee. A Steering Committee of eight (8) members, including the Chairman was chosen to guide the efforts of the standards group. In addition, about 30 other members were chosen to act in an advisory role in the Committee. Subsequent to this a number of other Committee meetings were held in conjunction with national UNIX-related conferences. These include:

- December 1981 at /usr/group meeting in Boston
- July 1982 at UNICOM conference in Boston
- August 1982 Systems Interface Sub-Committee meeting in Chicago
- November 1982 at COMDEX in Las Vegas
- January 1983 at UNICOM in San Diego

The major part of each of these meetings has been devoted to defining the System Interface Standard, referred to as /usr/group/standard.

### 4. Benefits

The potential benefits to all persons using the UNIX or functionally compatible operating systems are considerable for the following reasons:

- 1) Consumers will have a wider variety of products to choose from within the context of their own specific applications areas.

- 
- 2) Developers will be able to create products that can be sold in a much larger marketplace.
  - 3) Suppliers will enjoy an increased demand for their products because of a wider variety of applications making use of these products.
  - 4) Employers will find it easier to hire experienced, technically proficient staff; and the cost of training new or existing employees should be significantly reduced.
  - 5) Employees will enjoy greater job security because of a larger market for their skills and the reduced likelihood of technical obsolescence.
  - 6) Students will find the transition from theory to practice easier; instructors will find a larger selection of instructional materials available; and researchers will find it easier to exchange ideas.

The economic and educational benefits of the standard are well worth the effort of defining a standard at this time.

#### 5. Purpose

In the light of the considerable potential benefits to all members of /usr/group, the Standards Committee has spent a considerable amount of time trying to clearly define its purpose. This statement now reads:

"The purpose of the /usr/group Standards Committee is to formulate, adopt, publish and promote a formal standards specification, based on the UNIX operating system, for a commercial operating system. This standards specification is intended to assist persons producing or acquiring products based on the UNIX or functionally-compatible operating systems in accurately predicting the behavior of the products in the context of a specific implementation."

#### 6. Organization

The Committee determined at its initial meeting in December of 1981 that several issues were of immediate importance to the standard effort. Consequently, one temporary and three permanent Sub-Committee were formed so that the Committee could pursue the following goals simultaneously:

- 1) To identify the set of system calls and subroutines which will comprise the initial system interface standard; and to resolve any

---

ambiguities which are contained in the present definitions of these functions.

- 2) To identify a set of extensions to the system interface which are frequently required by commercial applications; and to define the operational characteristics of these functions.
- 3) To locate and/or define a viable standard for the "C" Programming Language.
- 4) To draft an organizational charter which outlines the purpose, benefits, costs, and feasibility of developing a viable commercial standard; and to draft a set of procedures by which the Standards Committee may conduct its proceedings and by which the /usr/group may formally adopt the standards developed by the Standards Committee.

The organization currently consists of eight Steering Committee members and about 25 advisory Committee members. The following are currently members of the Steering Committee.

|                   |                                              |
|-------------------|----------------------------------------------|
| Heinz Lycklama    | - Chairman                                   |
| Eric Petersen     | - Vice-Chairman & "C" Sub-Committee Chairman |
| Tom Hoffman       | - Secretary/Treasurer                        |
| Michael D. Tilson | - System Interface Sub-Committee Chairman    |
| John L. Bass      | - Extensions Sub-Committee Chairman          |
| Henry Burgess     | - Member                                     |
| Richard L. Ptak   | - Member                                     |
| Robert Swartz     | - Member                                     |

Steering Committee members have been chosen to insure a broad representation of producers, integrators, and end-users of the products which the /usr/group/standard is intended to benefit.

Advisory Committee members have been chosen based upon their expression of interest in the standardization activities. The current members of the Advisory Committee are:

|                |                 |                |
|----------------|-----------------|----------------|
| John Bass      | Jeff Goldberg   | Jeff Moskow    |
| Ross Bott      | Richard Hammons | Darwyn Peachey |
| David Buck     | Guy Harris      | Eric Petersen  |
| Henry Burgess  | Tom Hoffman     | Bill Plauser   |
| Pat Carruthers | Jim Isaak       | Richard Ptak   |
| David Clark    | Bill Joy        | Robert Swartz  |
| Brad Cox       | Don Kretsch     | Ted Tabloski   |
| Don Cragun     | Ben Laws        | Dean Thomas    |
| Craig Forney   | Heinz Lycklama  | Michael Tilson |
| Anthony Godino | Robert Michael  |                |

---

There are currently four active Sub-Committees. The purpose and status of each is discussed here.

#### 6.1 System Interface Sub-Committee

The initial system interface standard will be based on UNIX System III and will attempt to maintain compatibility with UNIX Version 7 wherever possible. The standard is intended to provide a basis for commercial applications development, and is an attempt to define a system interface based on the UNIX operating system. Special attention has been given to reducing or eliminating particularly machine-dependent functions.

Effort has concentrated on the original Sections 2 and 3 of the UNIX System III User's Manual. Section 2 defines the system calls which comprise the basic interface to the operating system. Section 3 defines the subroutines, both intrinsic and system call specific, that define the programming environment. Functions were omitted from the initial standard primarily because they either exhibited a high degree of machine dependency or were not usually required for applications development in the commercial marketplace. The Sub-Committee is presently editing manual sections based on UNIX System III and hopes to have a draft standard ready for publication early in 1983.

#### 6.2 Extensions Sub-Committee

The first major topic that was considered by the Extensions Sub-Committee was simply to determine what constituted an extension. After some discussion, it was generally agreed that extensions could include any commercially useful functions or facilities which were not described in the documentation distributed by Bell Laboratories, Western Electric, or AT&T. Extensions could, therefore, include any system calls, subroutines, utilities, or other programs which are not specifically documented in Volumes 1 and 2 of the UNIX Programmer's Manual.

Prior to the formation of the Extensions Sub-Committee, the Standards Committee as a whole had determined that the single most important technical subject that needed to be dealt with was the problem of controlling contention between multiple concurrent processes for shared data files ("record locking"). Three written proposals were considered from Dwayne Peachey of the Hospital Systems Study Group, John Bass of Fortune Systems, and E.J. McCauley of Zilog. After reviewing the various characteristics of each of these three proposals, the Sub-Committee decided to recommend adoption of the technique proposed by John Bass. Upon the Sub-Committee's recommendation, Mr. Bass agreed to clarify portions of the user

---

documentation and to provide a brief explanation of the technical rationale behind some of the more subtle characteristics of the technique. The description of this technique, including a suggested means of implementation, should be available for distribution to the /usr/group general membership early this year.

The Sub-Committee is actively soliciting written proposals from the /usr/group membership. Individuals wishing to make proposals to the Extensions Sub-Committee are encouraged to become members of the Standards Committee in order that they may present and explain their proposals in person. Individuals who are not members of the Standards Committee must find a member of the Committee to sponsor their proposal.

Written proposals should address issues of general interest to the /usr/group members and should document techniques or facilities which have already been implemented. The source code for implementing the technique should be in the public domain or the owner of its copyright must grant permission to the /usr/group Standards Committee to reproduce and distribute copies of the technique. Documentation or source code protected as proprietary information or trade secrets will not be accepted by the Standards Committee for consideration.

### 6.3 "C" Language Sub-Committee

In December of 1981, it was learned that Doug McIlroy of Bell Laboratories was planning to initiate an ANSI standardization effort for the "C" Programming Language. In March of 1982, Mr. McIlroy received consent from Bell Laboratories to proceed with this effort, although not as a representative of Bell Laboratories. In light of these facts, it was decided that the focus of the "C" Language Sub-Committee should be to maintain close contact with this ANSI standardization effort and to take an active role in the preparation and presentation of this standard to ANSI.

### 6.4 Organization and Procedures Sub-Committee

The organization and procedures temporary Sub-Committee has developed draft versions of the organizational Charter and By-Laws. These are in the process of being revised based on a review by members of the Standards Committee. The revised documents are being distributed to all members of the Standards Committee and the /usr/group Board of Directors. The intent is to have the Charter and By-Laws ratified by both the Standards Committee and the /usr/group Board of Directors early in 1983.

---

#### 6.4.1 Charter

The organizational Charter outlines the purpose, goals, scope, benefits, costs and feasibility of developing and implementing a viable standards. The Charter was written according to the format suggested by ANSI to facilitate a possible future submission of the standard to this body.

#### 6.4.2 By-Laws

The By-Laws describe the structure and procedures of the Standards Committee itself. Briefly, the Committee consists of eight Steering Committee members and up to thirty-two Advisory Committee members. Members of either Committee must be a general member of /usr/group.

#### 6.4.3 Steering Committee

Steering Committee members serve two-year staggered terms and are selected from the Advisory Committee by the Standards Committee Chairman, subject to the approval of other Steering Committee members. The Steering Committee is responsible for determining topics of discussion and for controlling publication of all standards documents.

#### 6.4.4 Advisory Committee

The Advisory Committee meets twice each year, at least, in conjunction with the regular /usr/group meetings. The primary criteria for membership in the Advisory Committee is interest, as evidenced by active participation in Standards Committee activities. Advisory Committee members may suggest topics for discussion and submit written proposals for inclusion in the standard. All draft standards will be published and distributed to members of the Advisory Committee for review and comment prior to being distributed to the /usr/group general membership. Advisory Committee members are nominated by the Chairman and approved by the Steering Committee.

### 7. Format

A long-term goal of the Standards Committee is to present the completed standards specification to the International Standards Organization (ISO) and the American National Standards Institute (ANSI) for adoption as an official ISO and/or ANSI Standard. Thus the format of the document will closely resemble others submitted to ANSI for adoption as a Standard.

The final format of the Standard is still under discussion and will probably include several sub-parts, some of which may be optional. The



---

following guidelines were used in editing the Draft Standards document, which is based on the UNIX System III User's Manual.

- a) The material will be presented in a style familiar to the UNIX programmer, but clearly identifiable as the /usr/group/standard.
- b) All machine-dependent references and facilities are to be removed.
- c) Many octal constants are translated into symbolic equivalents, using existing symbolic constants wherever possible.
- d) All constant values which imply limitations of magnitude and are implementation dependent are turned into separately identifiable symbolic constants, defined in a separate Appendix.
- e) All references to non-existent parts of the Standards document are removed.
- f) System call and subroutine names which are not part of the Standard will be put on a reserved name list.
- g) The Standard will be published along with a set of limits which must be specified by any Standard-conforming system.

The Standards document which is to be distributed for comments will actually come in two parts. The first part is the /usr/group/standard document itself. It consists of some introductory material plus edited manual pages derived from sections 2 and 3 of the UNIX System III User's Manual. The first section of the manual is essentially kept as a place holder for possible future standards efforts. Section 4 is also kept as a place holder. The other sections from the original UNIX documentation are combined into one section labelled as 'Miscellaneous'. This section contains only the edited manual pages required as referenced by manual pages in sections 2 and 3.

The second part of the document is a reviewer's guide. This is provided, as the name indicates, to provide the reviewer a roadmap to the /usr/group/standard document itself. The guide highlights the manual pages which have undergone major editing. Differences between the UNIX System III routines and the /usr/group/standard routines are pointed out as a guide for the reviewer. It also contains a list of system calls and subroutines that comprise the /usr/group/standard system interface. These are listed alongside the list of system calls and subroutines that appear in System III, Version 7 and 4.1BSD versions of the UNIX system for comparison purposes.

## 8. Procedures

The procedure for adopting standard consists of the following five basic steps:

- 1) A written proposal is submitted to or developed by the Steering Committee.
- 2) The Steering Committee Approves publication of a "DRAFT" standard which is then distributed to the Advisory Committee for review and comment.
- 3) The Steering Committee incorporates these comments into the standard and approves publication of a "DRAFT" standard which is then distributed to all general members of /usr/group for review and comment.
- 4) The Steering Committee incorporates these comments into the standard and approves publication of a "PROPOSED" standard, which is then distributed to all general members of /usr/group for voting by written ballot.
- 5) If two-thirds of the general members responding with written ballots are in favor of the "PROPOSED" standard, it becomes an "ADOPTED" standard and part of the official /usr/group/standard.

In summary, the process of adopting the standard involves soliciting comments from a group of individuals, revising the standard based on these comments, and distributing the revised standard to a larger group of individuals for further comment. Once a high degree of consensus has been achieved, a "PROPOSED" standard is published and distributed to the general membership of /usr/group for voting. If two thirds of the ballots returned are in favor of the proposed standard, it is considered formally adopted.

#### 9. Schedule

The current schedule for publishing the Standards Document is as follows:

- |   |            |                                                                                                                  |
|---|------------|------------------------------------------------------------------------------------------------------------------|
| 1 | Nov 30, 82 | COMDEX Meeting to discuss comments from Standards Committee and vendors.                                         |
| 2 | Jan 25, 83 | UNICOM Meeting to review DRAFT standards document.                                                               |
| 3 | Feb 28, 83 | DRAFT Standard Document made available to /usr/group office for distribution to interested parties for comments. |
| 4 | Jul 6, 83  | Last day for receiving comments on the DRAFT Standard document.                                                  |
| 5 | Aug 8, 83  | Publish PROPOSED Standard for Steering Committee review.                                                         |
| 6 | Sep 6, 83  | Publish PROPOSED Standard for /usr/group general membership vote.                                                |
| 7 | Oct 28, 83 | Deadline for receiving Ballots on PROPOSED Standard.                                                             |

---

8 Nov 7, 83      ADOPTED /usr/group/standard  
ready for distribution.

However, meeting this schedule assumes that the reviews are largely favorable and that there are no publication snags.

## 10. Legal Issues

The three most important legal issues that the Standards Committee continues to be concerned about involve copyright, trademark and antitrust law.

### 10.1 Copyright Issues

The material that we are using for editing the /usr/group/standard is the UNIX User's Manual for System III. This is copyrighted by Bell Telephone Laboratories. We have permission from AT&T to use selected pages from this manual for inclusion in documents published by /usr/group provided that the source of any such pages is acknowledged and any relevant copyright notice is reproduced.

### 10.2 Trademark Issues

UNIX is a trademark of Bell Laboratories. To avoid problems with AT&T we should use the "TM" trademark symbol after each occurrence of the word "UNIX". Any reference to the UNIX system must be a reference only to a system developed by Bell Laboratories and furnished pursuant to a license with Western Electric or AT&T. Any system furnished by an AT&T licensee cannot be called a UNIX system, but only derived from a UNIX system. The word "UNIX" can only be used to identify operating systems or other software whose source is Bell Laboratories. The Standards Committee must avoid using the word "UNIX" when referring to the /usr/group standard interface.

### 10.3 Antitrust and Unfair Competition Issues

The Standards Committee has taken precautions to avoid claims that use of the standard violates antitrust laws or the common law of unfair competition. There is no attempt to stabilize or fix the price of operating systems, to eliminate competition or to control production levels. The Standard is being adopted on a suggested basis and /usr/group members are not required to adhere to the /usr/group standard interface. /usr/group will not disparage or sanction a member or non-member who makes products which do not conform to the standard interface.

An attempt is being made to give all persons affected by the Standard, whether they be vendors, users or OEM's, an opportunity to participate

---

in the process of creating the Standard. As another precaution, /usr/group will not test products to determine the degree to which they conform to the proposed Standard.

---

# The History and Purpose of Standards

*Eric Petersen*

P2/I  
Suite 202  
5345 Wyoming Blvd. N.E.  
Abuquerque, NM 87109

This talk will present a brief history of the development of standards with emphasis on computer related standards. Characteristics of both successful and unsuccessful past standards efforts will be identified and discussed along with the general purpose of standards and their relationship to both consumers and suppliers.

## Standards Organization: Levels and Measurement

*Jim Isaak*

Charles River Data Systems  
4 Tech Circle  
Natick, MA

### *Why break out levels of implementation?*

- What is called "standard" and how does a developer and/or end user know what he is getting?
- Portable programs, portable operations, portable media
- Sub-sets (diskless, workstations)
- Environments *versus* operating systems
- Extensions: record locking, real time, keyed access files

### *How to tackle the task*

- Sort out the logical groups of routines (file management, process management). These define the "modules."
- For each group, identify levels. Can these be left out altogether and leave a meaningful subset? (Level 0 = Null) What does every program need? (Level 1 = Nucleus) What does the next level of capabilities require? (Level 2 = extensions) Are there more esoteric operations that few will require but are logically connected with this module? (Level 3 = options)
- As new elements are brought forward, they can be fit into existing modules, or defined as new ones.

### *Timing and commitment*

- By clearly setting this concept up as an objective, the standards group can start evaluating the approaches and the developers can start monitoring their products to know what module/levels they require or provide.
- Early commitment will also signal the possibility for subsets and encourage a wider spread level of compatible implementations.

---

*An example*

Modules and levels for the subroutines presented as part of the proposed standard (not complete, nor for discussion, but a few obvious examples).

*Measurement, presentation and validation*

The real payoff is in making it easy for a vendor to define what capabilities his products offer, and for an application developer to know what range of facilities to expect, or require for his environment.

And most importantly, the end-user can make a quick and straightforward evaluation of the systems available and applications available to make sure he gets what he needs.

Validation suites from University and/or governmental agencies will lend to a higher quality of "standard environments."

And ANSI standardization with NBS validation would be best approached with a multi-level multi-module standard that would span a wide range of systems and applications.

## Criteria for Standards

*Robert Swartz*

Mark Williams Co.  
1430 West Wrightwood  
Chicago, IL 60614

The operating system standard is one "which will assist persons producing or acquiring products based on these systems in accurately predicting the behavior of the product in the context of a specific implementation." (/usr/group Standards Committee Draft Proposal for Charter, December 10, 1981.)

What will be discussed is how to accomplish his goal. The criteria for inclusion of modules in the standard or their exclusion will be covered. The major criteria should be that the standard should include the smallest number of elements which support the largest number of user requirements.

---

**This page intentionally left blank**

## Permuted Index to Titles

|                                  |                                              |                                       |     |
|----------------------------------|----------------------------------------------|---------------------------------------|-----|
| .....                            | UNIX System III and .....                    | 4.1BSD                                | 25  |
| .....                            | Experiences in Porting .....                 | 4.1BSD UNIX to the $\lambda$ 750 VLSI | 132 |
| .....                            | Mail Systems and Addressing in .....         | 4.2BSD                                | 53  |
| .....                            | Windows with .....                           | 4.2BSD                                | 260 |
| .....                            | .....                                        | 4.2BSD on the Sun Workstation         | 132 |
| .....                            | UNIX for the Computer Automation .....       | 4/95                                  | 307 |
| .....                            | Licensing .....                              | Activity and Pricing                  | 49  |
| Environment .....                | Towards a UNIX System .....                  | Ada Programming Support               | 143 |
| .....                            | Mail Systems and .....                       | Addressing in 4.2BSD                  | 53  |
| .....                            | Interactive Data .....                       | Analysis using the Software Tools     | 8   |
| .....                            | Computer .....                               | Animation at UCSD                     | 261 |
| .....                            | UNIX .....                                   | APL                                   | 330 |
| .....                            | UNIX on .....                                | Apollo Computers                      | 133 |
| UNIX .....                       | .....                                        | Architectural Implications of         | 307 |
| UNIX-based .....                 | .....                                        | ARIEL: An Experimental                | 167 |
| Report .....                     | Rockies .....                                | Association for Tools (RAFT)          | 1   |
| .....                            | .....                                        | AT&T                                  | 23  |
| .....                            | Tools in .....                               | Australia                             | 14  |
| .....                            | UNIX for the Computer .....                  | Automation 4/95                       | 307 |
| .....                            | UC .....                                     | Berkeley                              | 23  |
| Retrieval .....                  | .....                                        | BIBFIND A Bibliographic               | 63  |
| .....                            | BIBFIND A .....                              | Bibliographic Retrieval System        | 63  |
| .....                            | Development of <i>refer.</i> .....           | Bug Fixes and Enhancements            | 99  |
| .....                            | RAPID: A Tool for .....                      | Building Interactive                  | 105 |
| .....                            | Software Tools in .....                      | C?                                    | 3   |
| .....                            | A Global Optimizing .....                    | C Compiler                            | 151 |
| .....                            | VMS .....                                    | C Compiler                            | 330 |
| .....                            | A Tutorial on .....                          | C Portability                         | 315 |
| .....                            | .....                                        | C Programming Environment             | 111 |
| .....                            | Ctrace - A Portable Debugger for .....       | C Programs                            | 187 |
| .....                            | An Implementation of the <i>fork</i> / ..... | Call for PDP-11 UNIX                  | 40  |
| .....                            | Getting Venture .....                        | Capital                               | 308 |
| .....                            | Meeting the Coming UNIX Training .....       | Challenge                             | 177 |
| <i>lex</i> and <i>yacc</i> ..... | .....                                        | COBOL Compiler Construction /         | 69  |
| .....                            | Meeting the .....                            | Coming UNIX Training Challenge        | 177 |
| .....                            | The Informix .....                           | Commercial DBMS for UNIX              | 245 |
| .....                            | /usr/group - Standards .....                 | Committee                             | 23  |
| .....                            | The /usr/group Standards .....               | Committee                             | 335 |
| .....                            | VAX11 .....                                  | Compatibility on PDP-11s              | 193 |
| System III .....                 | IS/3: A .....                                | Compatible Extension of UNIX          | 325 |
| .....                            | UNIX Markets and .....                       | Competition                           | 308 |
| .....                            | A Global Optimizing C .....                  | Compiler                              | 151 |
| .....                            | VMS C .....                                  | Compiler                              | 330 |
| <i>lex</i> and <i>yacc</i> ..... | COBOL .....                                  | Compiler Construction /               | 69  |
| .....                            | .....                                        | Computer Animation at UCSD            | 261 |
| .....                            | UNIX for the .....                           | Computer Automation 4/95              | 307 |
| .....                            | UNIX on Apollo .....                         | Computers                             | 133 |
| <i>lex</i> and <i>yacc</i> ..... | COBOL Compiler .....                         | Construction Experiences /            | 69  |
| .....                            | .....                                        | Contiguous Load Modules for UNIX      | 39  |
| .....                            | .....                                        | Criteria for Standards                | 349 |
| .....                            | .....                                        | CSNET Status Report                   | 51  |
| C Programs .....                 | .....                                        | Ctrace - A Portable Debugger /        | 187 |
| .....                            | Focus/USE: A Low Keystroke .....             | Database Editor                       | 241 |
| System .....                     | Design & Implementation of the DB .....      | Database Management                   | 211 |
| .....                            | Research .....                               | Database Management Software          | 201 |



|                     |                                        |                                              |     |
|---------------------|----------------------------------------|----------------------------------------------|-----|
| System .....        | Design & Implementation of the .....   | DB Database Management                       | 211 |
| .....               | The Informix Commercial .....          | DBMS for UNIX                                | 245 |
| .....               | Ctrace - A Portable .....              | Debugger for C Programs                      | 187 |
| .....               | .....                                  | DEC                                          | 24  |
| .....               | UNIX System .....                      | Definitions and Standards                    | 112 |
| Market .....        | .....                                  | Delivering UNIX to the End-User              | 311 |
| Enhancements ...    | .....                                  | Device Independent Graphics                  | 247 |
| .....               | Distribution and .....                 | Differentiation                              | 313 |
| .....               | Development of a .....                 | Digital Simulation System                    | 169 |
| .....               | The UNIX System: New .....             | Directions                                   | 109 |
| .....               | .....                                  | Distribution and Differentiation             | 313 |
| .....               | Writing User .....                     | Documentation for UNIX Systems               | 117 |
| .....               | Focus/USE: A Low Keystroke / .....     | Editor                                       | 241 |
| .....               | Delivering UNIX to the .....           | End-User Market                              | 311 |
| .....               | New UNIX Markets in .....              | Engineering                                  | 313 |
| .....               | Development of <i>refer.</i> / .....   | Enhancements                                 | 99  |
| .....               | Device Independent Graphics .....      | Enhancements                                 | 247 |
| .....               | .....                                  | Enhancements to <i>format</i>                | 7   |
| .....               | Hewlett-Packard's .....                | Entry into the UNIX Community                | 119 |
| .....               | .....                                  | EUNICE                                       | 284 |
| .....               | UNIX File System .....                 | Evolution                                    | 110 |
| Performance .....   | .....                                  | Evolution of UNIX System                     | 110 |
| UNIX .....          | .....                                  | Experiences in Porting 4.1BSD                | 132 |
| .....               | COBOL Compiler Construction .....      | Experiences Using <i>lex</i> and <i>yacc</i> | 69  |
| Interactive / ..... | ARIEL: An .....                        | Experimental UNIX-based                      | 167 |
| .....               | IS/3: A Compatible .....               | Extension of UNIX System III                 | 325 |
| <i>yacc</i> .....   | COBOL Compiler Construction .....      | Experiences Using <i>lex</i>                 | 69  |
| .....               | UNIX .....                             | File System Evolution                        | 110 |
| .....               | Development of <i>refer.</i> Bug ..... | Fixes and Enhancements                       | 99  |
| Database .....      | .....                                  | Focus/USE: A Low Keystroke                   | 241 |
| .....               | Enhancements to .....                  | <i>format</i>                                | 7   |
| Environment .....   | A .....                                | Friendly Text Processing                     | 116 |
| .....               | .....                                  | Getting Venture Capital                      | 308 |
| .....               | A .....                                | Global Optimizing C Compiler                 | 151 |
| .....               | The Port of UNIX to the .....          | Gould 32/27                                  | 273 |
| .....               | Device Independent .....               | Graphics Enhancements                        | 247 |
| Workstations .....  | .....                                  | Graphics Standards for Personal              | 257 |
| .....               | West Coast Implementors .....          | Group Proposed Standards                     | 15  |
| .....               | Users .....                            | Group Status Report                          | 1   |
| on a 16-Bit .....   | .....                                  | Handling Very Large Programs                 | 41  |
| the UNIX .....      | .....                                  | Hewlett-Packard's Entry into                 | 119 |
| .....               | The .....                              | History and Purpose of Standards             | 348 |
| .....               | A Compatible Extension of UNIX / ..... | III IS/3:                                    | 325 |
| .....               | UNIX System .....                      | III and 4.1BSD                               | 25  |
| .....               | Update on Software Tools .....         | Implementation                               | 14  |
| Management .....    | The Design & .....                     | Implementation of the DB Database            | 211 |
| System Call .....   | An .....                               | Implementation of the <i>vfork</i>           | 40  |
| Standards .....     | West Coast .....                       | Implementors Group Proposed                  | 15  |
| .....               | Architectural .....                    | Implications of UNIX                         | 307 |
| UNIX .....          | .....                                  | Improved Schedulers for Non-Paged            | 39  |
| Enhancements ...    | Device .....                           | Independent Graphics                         | 247 |
| UNIX .....          | The .....                              | Informix Commercial DBMS for                 | 245 |
| .....               | Interactive System/Three and the ..... | Intel                                        | 229 |
| .....               | RAPID: A Tool for Building .....       | Interactive                                  | 105 |
| Software .....      | .....                                  | Interactive Data Analysis                    | 8   |
| the Intel .....     | .....                                  | Interactive System/Three and                 | 229 |
| System .....        | ARIEL: An Experimental / .....         | Interactive Video Information                | 167 |
| .....               | A Uniform and Simple User .....        | Interface to UNIX                            | 113 |

|                                                    |                                   |     |
|----------------------------------------------------|-----------------------------------|-----|
| ..... The .....                                    | IS/1 Workbench for VAX/VMS        | 199 |
| UNIX .....                                         | IS/3: A Compatible Extension of   | 325 |
| ..... Focus/USE: A Low .....                       | Keystroke Database Editor         | 241 |
| ..... in Porting 4.1BSD UNIX to the .....          | λ750 VLSI                         | 132 |
| ..... The NIAL .....                               | Language Project                  | 331 |
| Super-micro .....                                  | Handling Very .....               | 41  |
| Security) .....                                    | LINUS .....                       | 143 |
| ..... Standards Organization: .....                | Levels and Measurement            | 348 |
| ..... .....                                        | Licensing Activity and Pricing    | 49  |
| UNIX .....                                         | LINUS (Leading Into Noticeable    | 143 |
| Tools .....                                        | LISP for the Software             | 15  |
| ..... Contiguous .....                             | Load Modules for UNIX             | 39  |
| ..... UNIX .....                                   | Logo                              | 145 |
| ..... REGULUS, a Real-Time UNIX .....              | Lookalike                         | 268 |
| ..... Focus/USE: A .....                           | Low Keystroke Database Editor     | 241 |
| ..... UNIX Research at .....                       | Lucasfilms                        | 167 |
| Tools .....                                        | A Portable .....                  | 7   |
| in 4.2BSD .....                                    | Mail System for the Software      | 53  |
| ..... Delivering UNIX to the End-User .....        | Mail Systems and Addressing       | 311 |
| ..... UNIX .....                                   | Market                            | 308 |
| ..... New UNIX .....                               | Markets and Competition           | 313 |
| ..... Standards Organization: Levels/ .....        | Markets in Engineering            | 348 |
| Challenge .....                                    | Measurement                       | 177 |
| System .....                                       | Meeting the Coming UNIX Training  | 116 |
| ..... UNIX Time-sharing .....                      | Menu-driven Office                | 279 |
| ..... A .....                                      | Menu-Driven Real-Time UNIX System | 24  |
| ..... Sun .....                                    | Microsystems                      | 39  |
| ..... Contiguous Load .....                        | Modules for UNIX                  | 269 |
| ..... UNIX for the .....                           | National 16032                    | 291 |
| ..... UNIX on the .....                            | National Semiconductor NS16032    | 51  |
| ..... The Plexus .....                             | Networked UNIX                    | 331 |
| ..... The .....                                    | NIAL Language Project             | 39  |
| ..... Improved Schedulers for .....                | Non-Paged UNIX Systems            | 143 |
| ..... LINUS (Leading Into .....                    | Noticeable UNIX Security)         | 291 |
| ..... UNIX on the National / .....                 | NS16032                           | 48  |
| ..... System V .....                               | Offering                          | 48  |
| ..... System V Support .....                       | Offering                          | 116 |
| ..... UNIX Time-sharing Menu-driven .....          | Office System                     | 333 |
| ..... SOLID: for .....                             | On-Line Systems Development       | 9   |
| ..... New Tools for the Virtual .....              | Operating System                  | 151 |
| ..... A Global .....                               | Optimizing C Compiler             | 348 |
| Measurement .....                                  | Standards .....                   | 40  |
| ..... An Implementation of the <i>fork</i> / ..... | Organization: Levels and          | 193 |
| ..... VAX11 Compatibility on .....                 | PDP-11 UNIX                       | 110 |
| ..... Evolution of UNIX System .....               | PDP-11s                           | 257 |
| ..... Graphics Standards for .....                 | Performance                       | 51  |
| ..... The .....                                    | Personal Workstations             | 251 |
| ..... Terminal-Independent .....                   | Plexus Networked UNIX             | 273 |
| ..... The .....                                    | Plotting Packages                 | 315 |
| ..... A Tutorial on C .....                        | Port of UNIX to the Gould 32/27   | 314 |
| ..... .....                                        | Portability                       | 187 |
| ..... Ctrace - A .....                             | Portability in the UNIX World     | 7   |
| Software .....                                     | Portable Debugger for C Programs  | 132 |
| VLSI .....                                         | Portable Mail System for the      | 285 |
| ..... Experiences in .....                         | Porting 4.1BSD UNIX to the λ750   | 49  |
| ..... .....                                        | Porting UNIX                      | 116 |
| ..... Licensing Activity and .....                 | Pricing                           | 111 |
| ..... A Friendly Text .....                        | Processing Environment            | 143 |
| ..... C .....                                      | Programming Environment           |     |
| ..... Towards a UNIX System Ada .....              | Programming Support Environment   |     |

|              |                                           |                                        |     |
|--------------|-------------------------------------------|----------------------------------------|-----|
| .....        | Ctrace - A Portable Debugger / .....      | Programs                               | 187 |
| .....        | Handling Very Large .....                 | Programs on a 16-Bit Super-micro       | 41  |
| .....        | West Coast Implementors Group .....       | Proposed Standards                     | 15  |
| .....        | The History and .....                     | Purpose of Standards                   | 348 |
| .....        | Rockies Association for Tools .....       | (RAFT) Report                          | 1   |
| Interactive  | .....                                     | RAPID: A Tool for Building             | 105 |
| Management   | .....                                     | /rdb: A Relational Data Base           | 237 |
| Lookalike    | REGULUS, a .....                          | Real-Time UNIX                         | 268 |
| .....        | A Menu-Driven .....                       | Real-Time UNIX System                  | 279 |
| .....        | Development of .....                      | refer. Bug Fixes and Enhancements      | 99  |
| Lookalike    | .....                                     | REGULUS, a Real-Time UNIX              | 268 |
| System       | /rdb: A .....                             | Relational Data Base Management        | 237 |
| .....        | UNIX .....                                | Research at Lucasfilms                 | 167 |
| Software     | .....                                     | Research Database Management           | 201 |
| .....        | BIBFIND A Bibliographic .....             | Retrieval System                       | 63  |
| (RAFT)       | .....                                     | Rockies Association for Tools          | 1   |
| .....        | UNIX on the National .....                | Semiconductor NS16032                  | 291 |
| .....        | LINUX (Leading Into Noticeable / .....    | Security)                              | 143 |
| .....        | A Uniform and .....                       | Simple User Interface to UNIX          | 113 |
| .....        | Development of a Digital .....            | Simulation System                      | 169 |
| .....        | Research Database Management .....        | Software                               | 201 |
| .....        | KEYNOTE ADDRESS: .....                    | Software Army on the March             | 17  |
| .....        | A Portable Mail System for the .....      | Software Tools                         | 7   |
| .....        | Interactive Data Analysis / .....         | Software Tools                         | 8   |
| .....        | Update on .....                           | Software Tools Implementation          | 14  |
| .....        | .....                                     | Software Tools in C?                   | 3   |
| .....        | LISP for the .....                        | Software Tools VOS                     | 15  |
| .....        | .....                                     | SOLID: for On-Line Systems Development | 333 |
| .....        | West Coast Implementors / .....           | Standards                              | 15  |
| .....        | UNIX System Definitions and .....         | Standards                              | 112 |
| .....        | The History and Purpose of .....          | Standards                              | 348 |
| .....        | Criteria for .....                        | Standards                              | 349 |
| .....        | /usr/group - .....                        | Standards Committee                    | 23  |
| .....        | The /usr/group .....                      | Standards Committee                    | 335 |
| Workstations | Graphics .....                            | Standards for Personal                 | 257 |
| Measurement  | .....                                     | Standards Organization: Levels         | 348 |
| .....        | Users Group .....                         | Status Report                          | 1   |
| .....        | CSNET .....                               | Status Report                          | 51  |
| .....        | UNIX for the .....                        | STD Bus                                | 185 |
| .....        | .....                                     | Sun Microsystems                       | 24  |
| .....        | 4.2BSD on the .....                       | Sun Workstation                        | 132 |
| .....        | Handling Very Large Programs / .....      | Super-micro                            | 41  |
| .....        | Towards a UNIX System Ada / .....         | Support Environment                    | 143 |
| .....        | System V .....                            | Support Offering                       | 48  |
| .....        | The UNIX .....                            | System: New Directions                 | 109 |
| .....        | Improved Schedulers for Non-Paged / ..... | Systems                                | 39  |
| .....        | Writing User Documentation for / .....    | Systems                                | 117 |
| .....        | Mail .....                                | Systems and Addressing in 4.2BSD       | 53  |
| .....        | SOLID: for On-Line .....                  | Systems Development                    | 333 |
| .....        | Interactive .....                         | System/Three and the Intel             | 229 |
| Packages     | .....                                     | Terminal-Independent Plotting          | 251 |
| .....        | A Friendly .....                          | Text Processing Environment            | 116 |
| Office       | UNIX .....                                | Time-sharing Menu-driven               | 116 |
| .....        | RAPID: A .....                            | Tool for Building Interactive          | 105 |
| .....        | A Portable Mail System/ .....             | Tools                                  | 7   |
| .....        | Interactive Data Analysis / .....         | Tools                                  | 8   |
| System       | New .....                                 | Tools for the Virtual Operating        | 9   |
| .....        | Update on Software .....                  | Tools Implementation                   | 14  |

|                                         |                                     |     |
|-----------------------------------------|-------------------------------------|-----|
| .....                                   | Tools in Australia                  | 14  |
| ..... Software                          | Tools in C?                         | 3   |
| ..... Rockies Association for           | Tools (RAFT) Report                 | 1   |
| ..... LISP for the Software             | Tools VOS                           | 15  |
| Programming/                            | Towards a UNIX System Ada           | 143 |
| ..... Meeting the Coming UNIX           | Training Challenge                  | 177 |
| ..... A                                 | Tutorial on C Portability           | 315 |
| ..... Computer Animation at             | UC Berkeley                         | 23  |
| .....                                   | UCSD                                | 261 |
| Interface                               | Uniform and Simple User to          | 113 |
| ..... A                                 | Update on Software Tools            | 14  |
| Implementation                          | UNIX                                | 285 |
| ..... Porting                           | UNIX for the Computer               | 307 |
| Automation                              | UNIX for the STD Bus                | 185 |
| .....                                   | UNIX Logo                           | 145 |
| .....                                   | UNIX on Apollo Computers            | 133 |
| .....                                   | UNIX System III and 4.1BSD          | 25  |
| Office                                  | UNIX Time-sharing Menu-driven       | 116 |
| ..... Contiguous Load Modules for       | UNIX                                | 39  |
| ..... The Plexus Networked              | UNIX                                | 51  |
| ..... A Uniform and Simple User /       | UNIX                                | 113 |
| ..... The Informix Commercial DBMS for  | UNIX                                | 245 |
| ..... Architectural Implications of     | UNIX                                | 307 |
| .....                                   | UNIX APL                            | 330 |
| ..... Hewlett-Packard's Entry into the  | UNIX Community                      | 119 |
| .....                                   | UNIX File System Evolution          | 110 |
| .....                                   | UNIX Markets and Competition        | 308 |
| ..... New                               | UNIX Markets in Engineering         | 313 |
| Semiconductor                           | UNIX on the National                | 291 |
| ..... LINUS (Leading Into Noticeable    | UNIX Security)                      | 143 |
| Support                                 | UNIX System Ada Programming         | 143 |
| Standards                               | UNIX System Definitions and         | 112 |
| ..... IS/3: A Compatible Extension of   | UNIX System III                     | 325 |
| ..... The                               | UNIX System: New Directions         | 109 |
| ..... Evolution of                      | UNIX System Performance             | 110 |
| ..... Improved Schedulers for Non-Paged | UNIX Systems                        | 39  |
| ..... Writing User Documentation for    | UNIX Systems                        | 117 |
| ..... Delivering                        | UNIX to the End-User Market         | 311 |
| ..... Experiences in Porting 4.1BSD     | UNIX to the $\lambda$ 750 VLSI      | 132 |
| ..... Portability in the                | UNIX World                          | 314 |
| .....                                   | UNIX for the National 16032         | 269 |
| ..... REGULUS, a Real-Time              | UNIX Lookalike                      | 268 |
| .....                                   | UNIX Research at Lucasfilms         | 167 |
| ..... A Menu-Driven Real-Time           | UNIX System                         | 279 |
| ..... The Port of                       | UNIX to the Gould 32/27             | 273 |
| ..... Meeting the Coming                | UNIX Training Challenge             | 177 |
| Video                                   | UNIX-based Interactive              | 167 |
| Improved                                | UNIX Systems                        | 39  |
| .....                                   | USENIX                              | 22  |
| Systems                                 | User Documentation for UNIX         | 117 |
| ..... Writing                           | User Interface to UNIX              | 113 |
| ..... A Uniform and Simple              | Users Group Status Report           | 1   |
| .....                                   | Using <i>lex</i> and <i>yacc</i>    | 69  |
| ..... COBOL Compiler Construction /     | using the Software Tools            | 8   |
| ..... Interactive Data Analysis         | /usr/group - Standards Committee    | 23  |
| .....                                   | /usr/group                          | 22  |
| ..... The                               | /usr/group Standards Committee      | 335 |
| UNIX                                    | <i>vfork</i> System Call for PDP-11 | 40  |
| ..... An Implementation of the          |                                     |     |

---

|                                             |                                |     |
|---------------------------------------------|--------------------------------|-----|
| .....                                       | VAX11 Compatibility on PDP-11s | 193 |
| ..... The IS/1 Workbench for .....          | VAX/VMS                        | 199 |
| ..... Getting .....                         | Venture Capital                | 308 |
| ..... An Experimental UNIX-based / .....    | Video Information System       | 167 |
| ..... New Tools for the .....               | Virtual Operating System       | 9   |
| ..... Experiences in Porting 4.1BSD / ..... | VLSI                           | 132 |
| .....                                       | VMS C Compiler                 | 330 |
| ..... LISP for the Software Tools .....     | VOS                            | 15  |
| .....                                       | Welcome                        | 17  |
| <b>Proposed</b> .....                       | West Coast Implementors Group  | 15  |
| .....                                       | Windows with 4.2BSD            | 260 |
| ..... The IS/1 .....                        | Workbench for VAX/VMS          | 199 |
| ..... 4.2BSD on the Sun .....               | Workstation                    | 132 |
| ..... Graphics Standards for Personal ..... | Workstations                   | 257 |
| <b>UNIX</b> .....                           | Writing User Documentation for | 117 |

---

## Author Index

- Allen, Bill**, REGULUS, a Real-Time UNIX Look-alike 268
- Allman, Eric**, Mail Systems and Addressing in 4.2BSD 53
- Appelbe, Bill**, *et al.*, Architectural Implications of UNIX 307
- Appelbe, Bill**, Welcome 17
- Berbers, Y.**, *et al.*, Porting UNIX 285
- Bishop, Mitch**, Handling Very Large Programs on a 16-Bit Super-micro 41
- Blackett, Kent**, A Menu-Driven Real-Time UNIX System 279
- Blevins, Jack**, The Port of UNIX to the Gould 32/27 273
- Britten, Chet**, *et al.*, Experiences in Porting 4.1BSD to the  $\lambda$ 750 VLSI 132
- Calland, Bob**, Enhancements to *format* 7
- Cassidy, James**, *et al.*, Development of a Digital Simulation System 169
- Cerofolini, Luigi**, UNIX for the STD Bus 185
- Chambers, John**, *et al.*, UNIX System III and 4.1BSD 25
- Chen, Paul**, *et al.*, Experiences in Porting 4.1BSD to the  $\lambda$ 750 VLSI 132
- Clegg, Frederick W.**, Hewlett-Packard's Entry into the UNIX Community 119
- Conant, Robert E.**, COBOL Compiler Construction Experiences Using *lex* and *yacc* 69
- Denny, Michael**, Delivering UNIX to the End-User Market 311
- Dickey, Matt**, *et al.*, Architectural Implications of UNIX 307
- Dolan, Charlie**, *et al.*, LISP for the Software Tools VOS 15
- Elahian, Camran**, New UNIX Markets in Engineering 313
- Evans, Steven R.**, Windows with 4.2BSD 260
- Guffy, W. R.**, System V Offering 48
- Haight, R.C.**, *et al.*, ARIEL: An Experimental UNIX-based Interactive Video Information System 167
- Harvey, Brian**, UNIX Logo 151
- Hausman, Paul**, Tools in Australia 14
- Henshaw, John**, Update on Software Tools Implementation 14
- Hidley, Greg**, Device Independent Graphics Enhancements 247
- Hosler, Jay R.**, Meeting the Coming UNIX Training Challenge 177
- Isaak, Jim**, Standards Organization: Levels and Measurement 348
- Isley, Larry K.**, Licensing Activity and Pricing 49
- Jacobson, Van**, Interactive Data Analysis Using the Software Tools 8
- Jacobson, Van**, *et al.*, New Tools for the Virtual Operating System 9
- Jacobson, Van**, *et al.*, West Coast Implementors Group Proposed Standards 15
- Jenkins, M.A.**, The NIAL Language Project 331
- Jollitz, Bill**, *et al.*, UNIX on the National Semiconductor NS16032 291
- Karels, Michael**, An Implementation of the *fork* System Call for PDP-11 UNIX 40
- Katslve, Bob**, UNIX Markets and Competition 308
- Kersten, Martin**, *et al.*, Focus/USE: A Low Keystroke Database Editor 241
- King, Laura L.**, The Informix Commercial DBMS for UNIX 245
- Knudsen, D.B.**, *et al.*, ARIEL: An Experimental UNIX-based Interactive Video Information System 167
- Kramer, Steven M.**, LINUS (Leading Into Noticeable UNIX Security) 143
- Lamb, J.Ell**, Towards A UNIX System Ada Programming Support Environment 143
- Lawson, Jim**, UNIX Research at Lucasfilms 167
- Levine, John R.**, Interactive System/Three and the Intel Data Base Processor 229
- Loomis, Jeff**, *et al.*, Computer Animation at UCSD 261
- Lycklama, Heinz**, The /usr/group Standards Committee 335

- 
- Lyon, Tom, et al.**, 4.2BSD on the Sun Workstation 132
- Mackay, Don**, Terminal-Independent Plotting Packages 251
- Manis, Rod**, /rdb: A Relational Data Base Management System 237
- Martin, Dave, et al.**, LISP for the Software Tools VOS 15
- Martin, Dave**, Users Group Status Report 1
- Martin, Marlene**, Distribution and Differentiation 313
- Mashey, John**, Software Army on the March 17
- Mashey, John R.**, SOLID: for On-Line Systems Development 333
- Mayer, Herbert G.**, COBOL Compiler Construction Experiences Using *lex* and *yacc* 69
- McGinness, Jim, et al.**, Architectural Implications of UNIX 307
- Meine, Bill**, Rockies Association for Tools (RAFT) Report 1
- Mercurio, Phill, et al.**, Computer Animation at UCSD 261
- Miller, Richard**, UNIX for the National 16032 269
- Moyer, James A.**, BIBFIND — A Bibliographic Retrieval System 63
- Neelands, Paul**, UNIX for the National 16032 269
- Neyer, James A.**, UNIX Time-Sharing Menu-driven Office System 116
- Noel, Greg, et al.**, Architectural Implications of UNIX 307
- O'Dell, Mike**, Portability in the UNIX World 314
- Peachey, Darwyn**, Improved Schedulers for Non-Paged UNIX Systems 39
- Petersen, Eric**, The History and Purpose of Standards 348
- Pickard, Monte**, The Plexus Networked UNIX 51
- Powers, George**, A Global Optimizing C Compiler 151
- Pozgal, Steve**, UNIX for the Computer Automation 4/95 307
- Quarterman, John, et al.**, UNIX System III and 4.1BSD 25
- Querido, Bob, et al.**, Architectural Implications of UNIX 307
- Raves, William, et al.**, Development of a Digital Simulation System 169
- Relley, G. Brendan**, CSNET Status Report 51
- Rugaber, Spencer**, A Uniform and Simple User Interface to UNIX 113
- Sandel, Dave**, System V Support Offering 48
- Scherrer, Phill**, Software Tools in C? 3
- Seeley, Donn**, VAX11 Compatibility of PDP-11s 193
- Shannon, Bill, et al.**, 4.2BSD on the Sun Workstation 132
- Shantz, Michael**, Graphics Standards for Personal Workstations 257
- Shewmake, David T.I., et al.**, RAPID: A Tool for Building Interactive Information Systems 105
- Shienbrood, Eric R., et al.**, UNIX on Apollo Computers 133
- Skinner, Glenn C., et al.**, UNIX on the National Semiconductor NS16032 291
- Soeder, Carl A., et al.**, UNIX on Apollo Computers 133
- Steffen, J.L.**, Ctrace — A Portable Debugger for C Programs 187
- Stitt, F. W.**, Research Database Management Software for UNIX-based Microcomputers 201
- Sturgess, Chris**, UNIX for the National 16032 269
- Sventek, Joe**, A Portable Mail System for the Software Tools 7
- Sventek, Joe, et al.**, West Coast Implementors Group Proposed Standards 15
- Swartz, Robert**, Criteria for Standards 349
- Thomas, Rebecca, et al.**, Writing User Documentation for UNIX Systems 117
- Tilson, Michael**, A Tutorial on C Portability 315
- Torcaso, William**, The IS/1 Workbench for VAX/VMS 199
- Upshaw, Bob, et al.**, New Tools for the Virtual Operating System 9
- Upshaw, Bob, et al.**, West Coast Implementors Group Proposed Standards 15
- Uter, Tom**, Welcome 17
- Verbaeten, P., et al.**, Porting UNIX 285
- Ward, James R., et al.**, UNIX on Apollo Computers 133

---

**Ward, Robert**, The Design and Implementation of the DB Relational Database Management System 211

**Wasserman, Anthony I., et al.**, Focus/USE: A Low Keystroke Database Editor 241

**Wasserman, Anthony I., et al.**, RAPID: A Tool for Building Interactive Information Systems 105

**Webb, Kincade N., et al.**, UNIX on Apollo Computers 133

**Wilder, Henry**, Getting Venture Capital 308

**Williams, Ellen**, EUNICE 284

**Wood, Jean**, VMS C Compiler 330

**Yao, Joseph**, UNIX APL 330

**Yates, Jean, et al.**, Writing User Documentation for UNIX Systems 117

**Zemon, Arthur**, A Friendly Text Processing Environment 116

**Zucker, Steve**, Contiguous Load Modules for UNIX 39

**Zucker, Steven**, IS/3: A Compatible Extension of UNIX System III 325



